MSc Computer Science Project Report

School of Computer Science and Information Systems

Birkbeck College

University of London

2008

Autonomous Agents and Emergent Behaviour

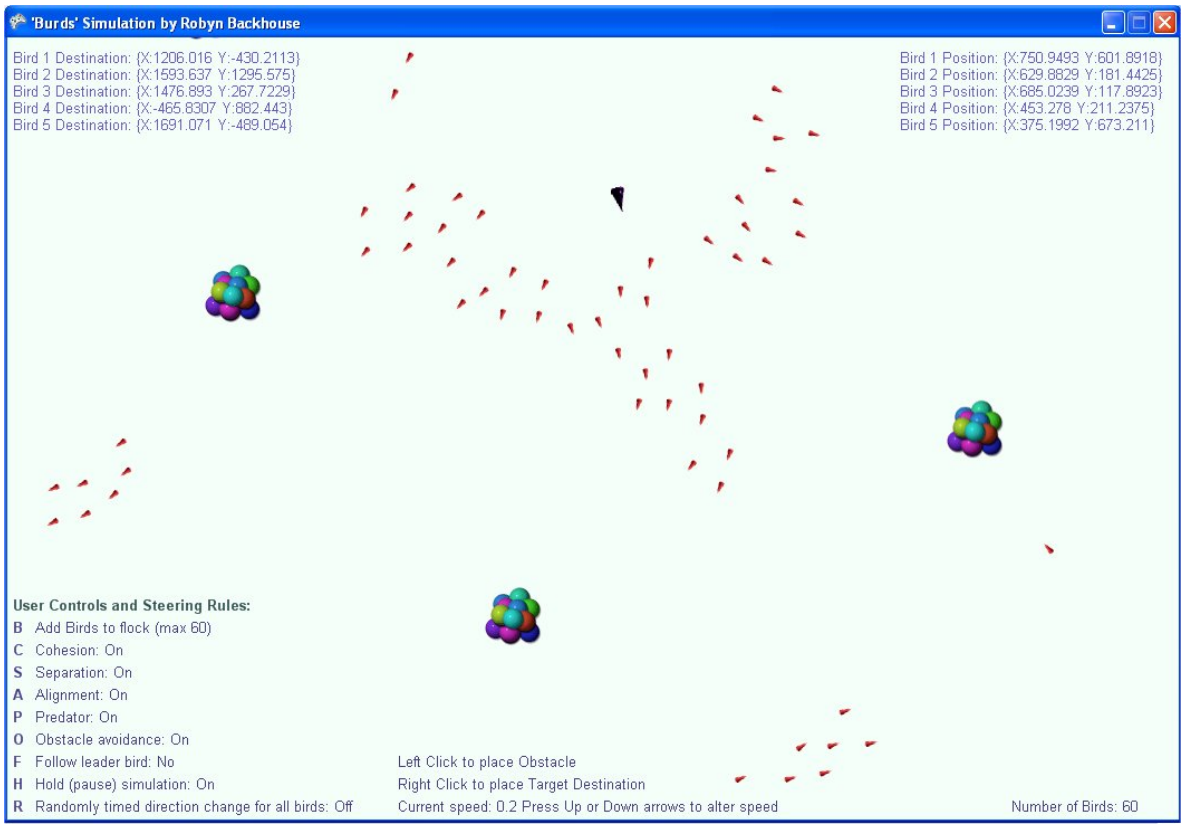
"Burds" Simulation

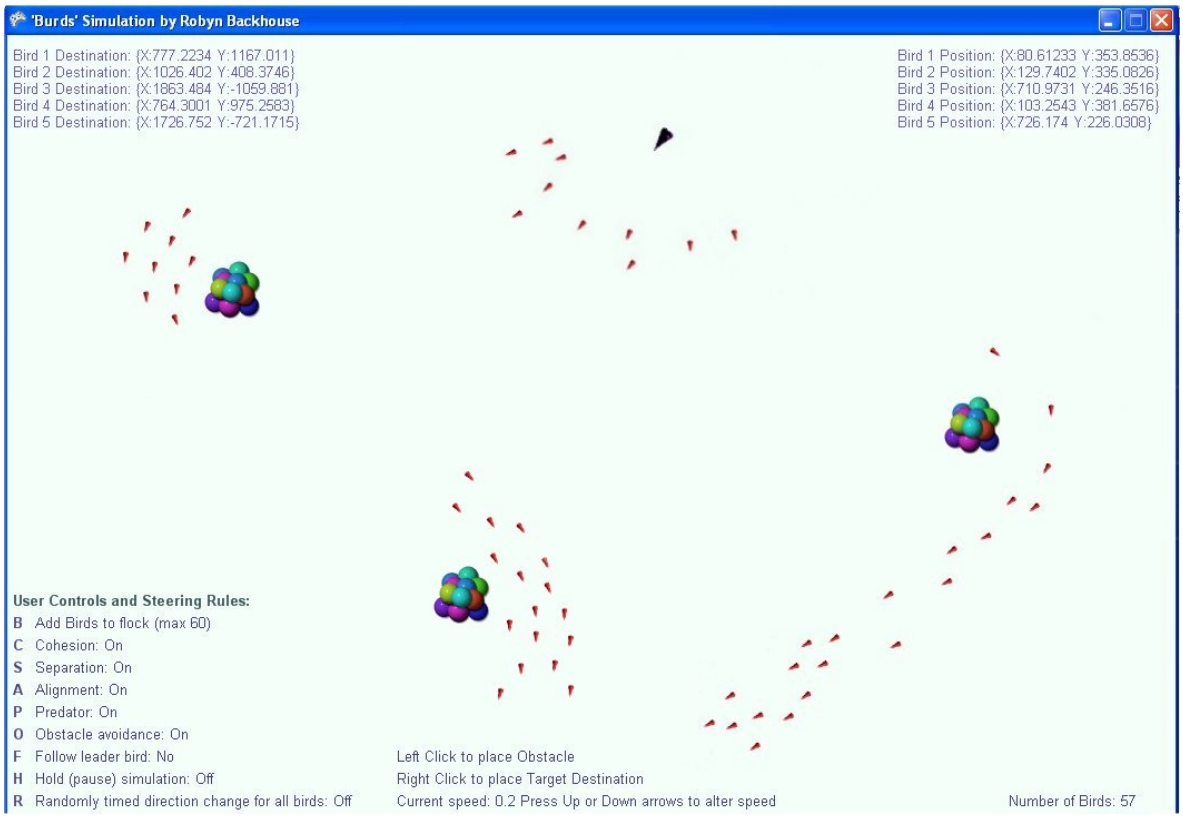
By Robyn Backhouse

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

TABLE OF CONTENTS:

APPENDICES:


'Burds' Simulation by Robyn Backhouse
Bird 1 Destination: {X:1206.016 Y:-430.2113}
Bird 2 Destination: {X:1593.637 Y:1295.575}
Bird 3 Destination: {X:1476.893 Y:267.7229}
Bird 4 Destination: {X:-465.8307 Y:882.443}
Bird 5 Destination: {X:1691.071 Y:-489.054}
Bird 1 Position: {X:750.9493 Y:601.8918}
Bird 2 Position: {X:629.8829 Y:181.4425}
Bird 3 Position: {X:685.0239 Y:117.8923}
Bird 4 Position: {X:453.278 Y:211.2375}
Bird 5 Position: {X:375.1992 Y:673.211}
User Controls and Steering Rules:
B Add Birds to flock (max 60)
C Cohesion: On
S Separation: On
A Alignment: On
P Predator: On
O Obstacle avoidance: On
F Follow leader bird: No
H Hold (pause) simulation: On
R Randomly timed direction change for all birds: Off
Left Click to place Obstacle
Right Click to place Target Destination
Current speed: 0.2 Press Up or Down arrows to alter speed
Number of Birds: 60


'Burds' Simulation by Robyn Backhouse
Bird 1 Destination: {X:777.2234 Y:1167.011}
Bird 2 Destination: {X:1026.402 Y:408.3746}
Bird 3 Destination: {X:1863.484 Y:-1059.881}
Bird 4 Destination: {X:764.3001 Y:975.2583}
Bird 5 Destination: {X:1726.752 Y:-721.1715}
Bird 1 Position: {X:80.61233 Y:353.8536}
Bird 2 Position: {X:129.7402 Y:335.0826}
Bird 3 Position: {X:710.9731 Y:246.3516}
Bird 4 Position: {X:103.2543 Y:381.6576}
Bird 5 Position: {X:726.174 Y:226.0308}
User Controls and Steering Rules:
B Add Birds to flock (max 60)
C Cohesion: On
S Separation: On
A Alignment: On
P Predator: On
O Obstacle avoidance: On
F Follow leader bird: No
H Hold (pause) simulation: Off
R Randomly timed direction change for all birds: Off
Left Click to place Obstacle
Right Click to place Target Destination
Current speed: 0.2 Press Up or Down arrows to alter speed
Number of Birds: 57

1: ABSTRACT:

Autonomous Agents are the subject of much research due to their myriad of potential uses notably in areas such as military weapons, medicine, graphics and communications. As research expands in this area it is essential that software can support these applications in the real world by simulating group behaviour effectively, allowing robots and agents to interact with their environment and each other unassisted by human intervention.

This simulation sets out to explore this area by emulating the natural flocking behaviour of birds requiring behavioural rules for cohesion, separation, alignment and avoidance of obstacles and predators. The birds effectively "think" for themselves and move in a self-organising manner allowing study of their emergent behaviour. A graphical user interface allows the user to select different combinations of rules, observing the resultant behavioural changes which occur. When all rules are selected the birds move around the screen in a cohesive flock avoiding obstacles, fleeing from predators, and for the most part avoiding collisions with each other.

The resulting simulation appears superior to many currently available, demonstrating improved collision avoidance between individual birds, and between birds and obstacles, both static and mobile. The simulation has also achieved all the original objectives set for the project. The next stage of development will be an adaptation from the current 2D implementation into 3D.

Supervisor: Giovanna Di Marzo Serugendo

2: INTRODUCTION AND MOTIVATIONS:

When deciding upon a project I had several requirements which I hoped would be fulfilled. I wanted to improve my programming skills and I wanted something which would be both challenging and interesting, consolidating and expanding the knowledge and skills learned during the MSc course providing me with a solid foundation upon which to build a future career in IT. I was also hoping to study an area of current interest and technological advancement

After consideration of several ideas I chose one suggested on the Birkbeck Computer Science website (Appendix C) studying autonomous agents and their emergent behaviour [MAN07] as it sounded very interesting, extremely challenging, and would certainly improve and greatly expand my programming skills as well as being an area of extensive current research and development.

I felt this would provide a great challenge and an excellent learning opportunity for further development. It presented the opportunity to learn more about graphics programming as well as a far greater affinity to the C# language specifically and object oriented programming in general. It also provided an opportunity to become more proficient at creating graphical user interfaces, as these are required for the vast majority of software applications and are therefore an essential programming skill.

Autonomous agents are currently being heavily researched across a wide range of fields and it is likely that commercial production of such systems will increase rapidly in the near future as systems currently under development progress to the point where they are ready for implementation within the real world in area's for which they are being designed. This project provided the opportunity to learn about how autonomous agents work and to gain a greater understanding of their capabilities, functionality and practical applications.

There are numerous types of autonomous agents being developed each working in different ways but which are all based on collections of individual units working in unison with a high level of autonomy. These include Complex Adaptive Systems, Population-based Adaptive Systems, Swarm Intelligence, Swarm Engineering, Multi-Agent Systems and Self Organising Systems [DAV05]. This project is a study of Swarm Intelligence.

2.1 Applications of Autonomous Agents using Swarm Intelligence:

Artificial intelligence is becoming more popular within military, scientific, industrial and commercial arenas as researchers strive to develop more autonomous robots for use in a wide range of applications and environments. Uses for autonomous agents are many and varied and are

increasing dramatically as technology advances. Some areas in which autonomous agents are used include games and movies, computer graphics and artificial life and even art, but the major use of Swarm Intelligence based autonomous agents is in the field of intelligent robotics [DAV05].

One type of autonomous agent is the swarm robot which is being heavily researched and developed due to the potential military and scientific uses. The popular press often reports on development of military swarms that will in future be sent into warfare or reconnaissance situations without the need for continuous direct human control such as the current ones require. Currently reported examples include "insect like" robots resembling things such as spiders, dragonflies and even worms which can carry out reconnaissance, communicate conditions to human troops, identify and monitor hostile targets, and even carry weapons [GIL08, USA08].

Another area in which such swarms would be invaluable is exploration as they will be capable of going into areas which are inhospitable to humans such as space, deep sea or radioactive areas. Even in rescue operations swarms may one day be used with large numbers of tiny robots searching for survivors of earthquakes or collapsed buildings etc., possibly even forming "teams" which can work together to move rubble [GAR08].

Swarms of small robots have several advantages over single large "humanoid" type robots: they are far cheaper with a whole "swarm" potentially costing less than a single large robot, they can cover large areas due to their numbers, and if one is destroyed or malfunctions the remaining ones simply rearrange themselves with minimal or no loss of overall function of the swarm [PAL08, GIL08].

Medicine is another important area for future swarm robots with hopes that swarms of "nanobots" or "nanites" will be capable of circulating in peoples bloodstreams attacking cancer cells or cleansing poisons and toxins from the blood [WIK08b]. Some authors even go so far as to predict that these nanites will one day be capable of such efficiency that humans will be able to breath toxic gases or fall from "10 storey buildings" without permanent injury, change sex at will, no longer suffer viral infections, pain or bruising, and that even the aging process itself may be halted [BRO00]. Perhaps closer to becoming reality within the immediate future are much larger robots working around hospitals in teams, communicating with each other and staff - locating and notifying doctors or other staff as they are needed, providing almost instant communication between staff throughout the hospital via voice and video links, cleaning, assisting visitors, and monitoring for such events as safety hazards (such as fluid on floors or obstructions) and patients falling [FRA07].

Movies and games commonly contain artificially created flocks or herds of animals, birds, fish, insects, or even humanoids using groups of autonomous agents. The first movie to use this type of

flocking behaviour was Batman Returns (swarming bats and armies of penguins) with Craig Reynolds pioneering and implementing this approach [MIL07]. Others soon followed and swarm behaviour is now commonly seen in movies such as The Matrix Revolutions (swarms of insect-like robots forming an animated face), The Lion King (herds of wildebeest) and Finding Nemo (schools of fish). Many modern games now boast stunning visuals which include flocks of birds, schools of fish and other lifelike non-playable characters, many of which interact with each other in very realistic ways often fighting and killing each other if their paths cross. Some particularly beautiful current examples of these include World of Warcraft and Guild Wars.

Swarms are even being used by artists creating interactive art using swarm intelligence where a "swarm" follows a users movements on a screen and dynamic artworks are created by the movements of swarming objects [BOY04, JAC07, JAC06].

Other potential uses seem endless and include areas such as routing of telecommunication networks, planetary mapping, controlling unmanned vehicles (e.g. for exploration, warfare or mining), data mining, power-grid controllers and interferometry [WIK08a, REY99].

2.2 Aims and Objectives:

In nature it is wonderful to observe groups of animals moving in unison such as schools of fish, herds of buffalo or flocks of birds. A flock of birds moves as a single entity flowing gracefully as if with a single consciousness yet clearly it is comprised of many individuals each with it's own characteristics [REY87]. As there is no known telepathic link between these individuals it is assumed they must base their movement within a group on a set of behaviours in order for the group as a whole to function cohesively. In nature these behaviours would depend on the creatures "senses" such as sight, sound and touch and would be instinctive and natural rather than deliberate and planned. In order to simulate this on a computer, rules and algorithms must be designed which artificially mimic these natural behaviours.

The main aim of this project was to create a flock of autonomous "characters" (in this case birds) which move around in a self-organising manner, and to provide them with a set of steering rules mimicking the natural movement of birds so their emergent behaviour may be observed when under the influence of various combinations of some or all of these rules. Additional secondary objectives were the addition of obstacle avoidance and finally predators into the simulation if time permitted.

Looking at existing simulations it is evident that some give a very natural looking flocking motion while others fall considerably short of this goal giving jerky or mechanical un-natural looking

movement, and in many cases having poor anti collision control resulting in birds continually moving into each other and colliding. The vision was to develop a simulation which combined the best features of existing simulations while hopefully improving upon the weaker areas such as collision avoidance and non-fluid motion.

Due to a complete lack of any prior graphics programming experience it was decided to implement the model primarily in 2D. The 2D implementation certainly appears more popular with other authors and allowed more focus to be put on the movement and behaviours themselves rather than on the complex modelling of 3D environments and drawing. This was more in keeping with the original aim of the project which was to explore emergent behaviour of autonomous agents rather than study 3D graphics and game programming.

Several elements required in order to achieve these objectives included development of a model for the simulation of the birds' behaviour using algorithms for steering and behaviour rules, production of a graphical user interface permitting the user to influence the birds behaviour, and a visualisation of the simulation.

3: RELATED WORKS:

There is a great deal of published literature on swarm intelligence and flock behaviour from many authors, most of whom appear to have based their work on that of Craig Reynolds. As the foremost authority on steering behaviour for autonomous characters (a type of autonomous agent used in computer animation such as games and virtual reality) Reynolds may be seen as the "father" of the flocking model introducing his original Boids model in 1987 which was a groundbreaking innovation at that time, being the first well known successful implementation of a natural looking flocking system, and which has been used extensively ever since particularly in areas such as visualisations in games and movies [REY01, REY99, REY88, REY87].

His model describes behaviour in a three layer hierarchy:

- Action Selection - strategy, goals and planning
- Steering - path determination
- Locomotion - animation, articulation [REY99]

Action Selection is defined as "noticing that the state of the world has changed and setting a goal" [REY99]. Steering is the art of selecting a path by which to reach the goal, turning appropriately, avoiding collisions and obstacles etc., and Locomotion is the actual act of moving along the path to reach the goal or final destination [REY99].

The focus of this project is predominantly the middle layer - steering, which is itself composed of several rules which the autonomous agent must obey in order to interact appropriately with the world and with other autonomous agents. Reynolds' original Boids model contained three steering rules: separation, cohesion and alignment, with further behaviours being added in subsequent years including obstacle avoidance and flee behaviour (e.g. from a predator). Avoidance of obstacles is only necessary when the obstacle lies directly in the birds' path (a bird may fly parallel to a wall without difficulty), whereas flee causes the bird to move away from the predator regardless of it's position in relation to the bird as long as it is within a certain distance.

Reynolds goes on to describe how some behaviours are inherently combined (e.g. fleeing from a predator while simultaneously avoiding obstacles), while others are not (e.g. stopping to eat while fleeing from a predator) [REY99].

Many authors have been inspired by Reynolds' model using it as a basis for their own work, some of which are presented briefly below. Parker [PAR07] produced pseudo code based directly upon Reynolds' model, while Davison [DAV05] discussed a Java implementation of the model in his book

Killer Game Programming. Both were referred to during development of the cohesion, separation and alignment rules for this project.

There appear to be numerous examples of flock simulations available and many have been studied to discern the most desirable attributes for inclusion into this simulation. Apart from Reynolds' 3D java applet simulation [REY01] two of the earliest simulations viewed (and therefore used as a starting point for planning this simulation) were those of Buckland [BUC05] and Bourg and Seemann [BOU04]. Buckland's simulation appeared to have no collision avoidance and no obstacles or predators, with the only user control being an option to show a neighbourhood circle around a bird. The flock movement however was flowing and natural looking and as one of the earliest simulations seen it formed an initial idea upon which the graphical representation of this project was based. It used 2D triangles to represent the birds which is what has been used in this simulation and indeed many others viewed since. Bourg and Seemann's simulation had very good obstacle avoidance with the birds avoiding obstacles and continuing in the same general direction rather than bouncing back from them as with most other simulations, however it is almost impossible to discern whether or not collisions occur between individual birds due to the graphical representation and the speed of movement.

Most authors who have implemented obstacle avoidance appear to have birds avoiding obstacles by selecting a different destination rather than continuing on their original path, and indeed when natural birds are observed this may often be seen. LaLena [LAL08] and Grubb [GRU07] both used this course altering behaviour when obstacles were encountered and this method was the one implemented into this simulation.

Grubb's [GRU07] user interface appeared well designed, again providing idea's which were incorporated into this simulation. He included the ability to enable and disable all steering rules individually, allowing the user to experiment with different combinations of rules to explore the resultant behaviour. It is very interesting to observe this altered behaviour and makes checking effectiveness of the rules simple - if only one rule is on at a time it is quite obvious whether or not it is working effectively. This was seen as an essential addition to my simulation and was implemented quite early.

Most simulations appear to have very poor or often non-existent collision avoidance (with other birds). Grubb's [GRU07], LaLena's [LAL08], Buckland's [BUC05], and Wiley's [WIL99, WIL99a, WIL99b] characters all continually collide, often moving permanently on top of each other. This was seen as highly undesirable and much work went into trying to overcome this problem for my simulation. The original algorithm did not work sufficiently well so was modified until acceptable results were obtained. It is now uncommon for birds to collide unless they are crowded from

multiple sides simultaneously while avoiding the predator in which case occasional collisions occur, but these are instantly rectified. Unlike the above simulations my birds do not "travel" on top of one another.

Most simulations viewed appear to have smooth flowing motion apart from some of Wiley's [WIL99] which do not appear to flock at all but rather flash random groups of characters around the screen. The way in which the rules control the birds appear to produce this flowing motion in most simulations with large jerky movements being prevented by the proximity and motion of the near neighbours. Another of Wiley's simulations [WIL99b] claims to run much faster than any others by using a different way of checking neighbours. He reports that instead of checking all birds in the flock to decide which ones are in the immediate neighbourhood it checks only the ones which were within the neighbourhood distance the previous time, along with their neighbours. This would certainly increase speed of the simulation however it may mean that some neighbours are missed and therefore not included in calculations, particularly if birds suddenly changed direction. It would seem that if new birds joined the flock they would not be considered as they would not previously have been neighbours to any existing birds.

Richmond [RIC07] presents a visually stunning 3D simulation of birds flocking in mountains with a camera slowly rotating around the full panorama and the birds scaling nicely in size as they move nearer or further from the viewer. They seem to disappear permanently after the simulation runs for some time although they may still be heard. The flock moves slowly around and stays together in a realistic looking way although it is difficult to see whether or not they collide due to them appearing so distant and small. Again some basic user controls are present although these do not appear to make much difference to the birds' behaviour and although there are predator controls, no predator is apparent. If an advanced 3D version is to be implemented in future the visualisation of Richmond's would indeed be something to aspire towards, although better camera control and user options would be suggested as would obstacles and an obvious predator.

Reynolds also presents a very simple 3D version on his website [REY01] which has no obstacles or predators, but which does appear to have very good anti-collision control.

4: BEHAVIOURAL RULES AND MODEL:

4.1 General Model Overview:

There are several methods by which a flock of birds, or indeed any group of creatures moving together such as schools of fish or herds of wildebeest, may be modelled. To create a realistic looking flock of birds it would be possible to hard code the flight path of every bird individually however this would be very difficult to update (for instance if the flight path was to be altered), and it would also be difficult to ensure that collisions never occurred as the flock turned and changed direction [REY87]. It also has the limitation of the finite length of hard coding – what happens to the birds once they have flown their set path? They may simply repeat their flight path in an endless loop however this would look very artificial and would be limited to a set area of the world not taking into account other environmental factors within their vicinity. If the simulation was being used in a game for instance, the flock would be local to just one area of the world and each area would have to have it's own individual hard coded flock. When thinking of real world implementations for autonomous agents this would be completely unworkable as it removes their autonomy and their ability to function within a changing environment.

In order to create a model which is more accurate, realistic looking, easily updated and mobile within various settings it is better to give autonomy to each bird so that it can make it's own "decisions" based on it's surroundings and immediate environment [REY87]. Each bird follows a set of behaviour rules by which it can interact with other birds in a similar way to that in which a real bird might interact. This obviously gives much greater flexibility to the flock as it can then fly anywhere using the same set of rules. If it is used in a game the same flock can fly in virtually any area in a natural looking way without being specifically hard coded to that environment. Similarly when considering the broader spectrum of uses for autonomous agents such as robots or weapons their ability to adapt themselves to different environments is highly desirable if not essential.

This model is based upon Craig Reynolds' flocking model whereby in order to simulate a flock those behavioural characteristics which are relevant to a bird being a successful member of a flock are simulated [REY87]. However the more autonomous the birds become the more difficult it becomes to directly control their movements. Reynolds reported that not being able to foresee how a simulation will react to set behaviours and conditions is "one of the charming aspects" [REY87]. Birds need to have certain behaviours in order to flock effectively. The two main behaviours required may be seen as opposing – those of flying closely together in a flock while simultaneously avoiding collisions with other birds. Drawing together, but repulsing apart at the same time.

In order to create a flocking model in this way three main rules are required:

- Cohesion: Birds try to move towards the centre of their group of immediate neighbours.

- Separation: Birds must not hit each other
- Alignment: Birds move towards the average destination of all their neighbours.

In addition to these basic rules for flight, if obstacles and/or predators are being introduced to the model then two further rules are required:

- Obstacle Avoidance: Birds must avoid running into obstacles
- Predator Avoidance: Birds must avoid being eaten by predators

4.1.1 Neighbourhood:

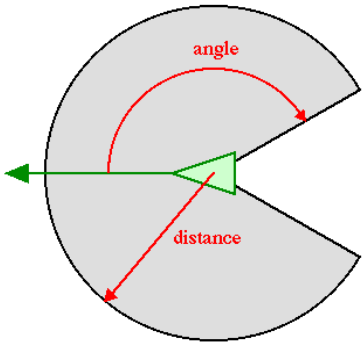


Diagram from [REY99]

Each birds perception of the flock is quite local to itself. For instance in nature a flock may number several hundred or even several thousand. It would be virtually impossible for an individual bird to consider every single flock member continuously. Instead it simply considers those immediately surrounding it – it's "nearest neighbours". Using this approach the rest of the flock outside it's immediate neighbours can be effectively ignored by each individual bird but because every bird follows the same behaviour the flock as a whole "works", regardless of its size. In this simulation therefore, a neighbourhood distance is set and any bird outside that radius is effectively ignored by each individual bird.

Neighbourhood distance is not only defined as a set distance from the bird but also within a set angle to that which the bird is currently facing so that a bird behind it will be ignored. This is achieved by calculating the angle between the direction in which the bird is facing and the angle at which the second bird is positioned in relation to the direction of the original bird. If the second bird is "within sight" of the first bird (i.e. within the grey area on the diagram) then it is considered to be within the neighbourhood. If it is in the "blind spot" of the first bird (i.e. behind it) it is ignored.

4.1.2 Division by Zero Prevention:

Checks are made to ensure that co-ordinates used as a denominator when carrying out vector division are not zero otherwise an exception will occur. In this simulation if they are found to be zero they are changed to the value of 0.1. All such calculations use float values. Although this may be unacceptable in critical systems in which even the slightest change may prove fatal to a system, for the purpose of this graphical simulation this figure is so small as to have no noticeable effect on the birds' destinations or positions, being imperceptible to the human eye, and was therefore acceptable.

4.2 Behaviour Rules in Detail:

The first three rules are those observed by Craig Reynolds [REY01] which are cohesion, separation and alignment, followed by obstacle avoidance and predator avoidance.

Each rule is weighted according to its level of importance. It was felt that separation (not bumping into each other) for instance was more important than cohesion or alignment so separation was given a much higher weight than the others. Avoidance of both obstacles and predators also has a higher weight than cohesion or alignment as it is assumed that to a real bird not flying into obstacles and potentially becoming injured, or indeed being eaten by a predator, are probably of far higher immediate priority than staying with other birds.

Each bird has a set of position co-ordinates (x, y) and a set of destination co-ordinates (x, y). The bird moves fractionally from its current position towards its destination each time it is "updated" which occurs approximately 60 times per second for each bird. Each time a birds co-ordinates are updated it is redrawn on screen in it's new position.

When a bird moves beyond the "edge" of the window it is wrapped around to the other side as if the window is an opened out map of the earth (like a globe flattened to a 2D map).

4.2.1 Update Bird's Position and Destination:

The procedure which updates the position and destination of each bird calls all behaviour rules which are currently selected and adds the resulting vectors together, along with the birds current destination and position. Each rule returns a vector of how much the birds destination should be changed by that particular rule, so simply adding all of these to the birds' current destination very effectively points the bird to its new corrected destination.

To update a bird's destination the current destination is added to the sum of all the steering rules which are currently selected. So for instance a bird has a destination of x and y co-ordinates. Each steering rule gives a new set of x and y co-ordinates which are added to the bird's current destination vector co-ordinates, so the new destination is simply the sum of the old destination and the values returned from all the behavioural rules which are selected at that time.

To find the new position the bird is moved a fraction of the distance between it's current position and it's newly calculated destination.

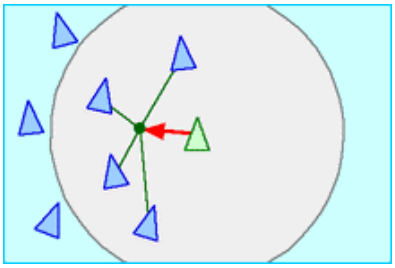
Algorithm : Update Position of Bird

```
1.  procedure NEWPOSITION
2.      for every bird in flock
3.          Crule = vector returned from cohesion rule
4.          Srule = vector returned from separation rule
5.          Arule = vector returned from alignment rule
6.          Orule = vector returned from obstacle avoidance rule
7.          Prule = vector returned from predator avoidance rule
8.          oldDest = current destination co-ordinates of bird
9.          oldPos = current vector co-ordinates of bird
10.         newDest = new updated destination co-ordinates of bird (initially NULL)
11.         newPos = new updated position co-ordinates of bird (initially NULL)

12.         newDest = oldDest + Crule + Srule + Arule + Orule + Prule
13.         newPos = oldPos + (fraction of)newDest
14.     end for
15. end procedure
```

The final destination vector is put through a “distance inhibitor function” which “normalises” the vector prior to the new position being calculated. In order to control the speed of the birds the new destination and new position vectors are further controlled by speed limiting functions and controllers (speed and distance inhibitors discussed later).

4.2.2 Cohesion Rule

The Cohesion rule brings the birds together into a flock or group. The birds try to move towards the centre of their group of immediate neighbours.



(Diagram from Craig Reynolds [REY01])

Calculation is straightforward. The positions of all neighbouring birds are added together then divided by the number of birds within that neighbourhood, giving the average position of all birds within that neighbourhood.

The centre of a birds neighbourhood may be calculated by the following algorithm:

Centre of neighbourhood = sum of all bird positions / number of birds in neighbourhood

The neighbourhood centre is then multiplied by the weighting for the cohesion rule before being returned and added to the bird's current destination (along with the other rules) so the new destination may be calculated.

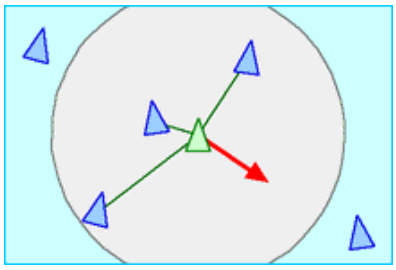
The code for this rule was originally based on pseudo code by Conrad Parker [PAR07] but after some experimentation it was found to give unacceptable results in this simulation so it was changed considerably and now appears to give consistently reliable results.

Algorithm : Cohesion Rule

1. Cohesion_ vector = new vector (0, 0)
2. bird1 = original bird
3. bird2 = 2nd bird being compared to original bird1
4. **while** more birds in flock
5. **if** bird2 is within set neighbourhood distance to bird1
6. Add bird2 position to Cohesion_vector
7. **end if**
8. **end while**
9. **if** number of neighbours > 0
10. **return** (Cohesion_vector / number of birds in neighbourhood) – position of bird1
11. **else return** vector of value (0, 0)
12. **end if else**

4.2.3 Separation Rule

Birds must avoid bumping into each other as they move maintaining a minimum distance between themselves and their neighbours. (also referred to as "collision avoidance").



(Diagram from Craig Reynolds [REY01])

The bird's distance from every other bird in the flock is calculated individually (the Euclidean distance) and if they are found to be closer to any other bird than the set minimum "collision distance" (i.e. the minimum distance which they should keep between them) they must move apart.

A separation vector is created with co-ordinates (0, 0). If birds are found to be too close together the separation vector is updated to equal its current value minus the position difference between the two birds. This is repeated for every bird in the flock which is closer than this minimum distance. The separation vector is then returned and added to the new destination for the bird. The pseudo code for the separation rule was based on pseudo code by Conrad Parker [PAR07].

For this rule the neighbourhood distance is not used as it would result in reduced performance by simply creating a redundant check due to the structure of the bird flock which is an array. It was felt that as every bird in the flock had to be tested for distance from the original bird in order to ascertain whether they were within the neighbourhood distance then it would be more efficient to simply check directly for collision distance rather than checking first for neighbourhood distance then rechecking those a second time for collision distance if they were within the neighbourhood distance. In real life birds would only consider those in their immediate vicinity and this would be done by direct visualisation, however a computer simulation using an array must check every bird in the flock in order to ascertain it's position in relation to the original bird.

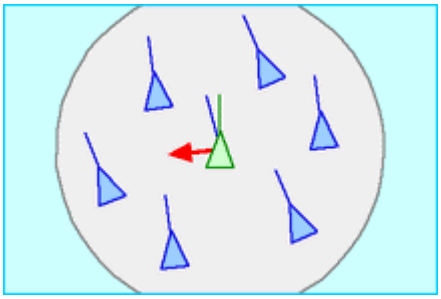
Again a weight is used as a multiplier for the final result, so that this rule may be given a higher importance in the final destination vector than other rules as it was felt that the most important thing while flying is not to hit anything – either other birds or obstacles.

Algorithm : Separation Rule

1. Separation_vector = new vector (0, 0)
2. bird1 = original bird
3. bird2 = 2nd bird being compared to original bird1
4. **while** more birds in flock
5. **if** bird2 is within collision distance to bird1
6. Separation_vector = Separation_vector – (position of bird2 – position of bird1)
7. **end if**
8. **end while**
9. **return** Separation_vector * weight

4.2.4 Alignment Rule

Birds move towards the average destination of their neighbours keeping the flock in alignment and moving together towards the same general heading.



(Diagram from Craig Reynolds [REY01])

The alignment rule calculates the average destination of all birds within a set neighbourhood distance from the original bird (including the destination of the original bird) and returns the average of those destinations.

The destinations of all birds within the neighbourhood distance from the original bird are added together and divided by the number of birds within that neighbourhood. The destination of the original bird is then subtracted from this averaged destination and the result divided by 8 (to add approximately an 8^{th} of the resultant vector to the birds current destination [PAR07]). This result is then multiplied by the weight for this rule and returned.

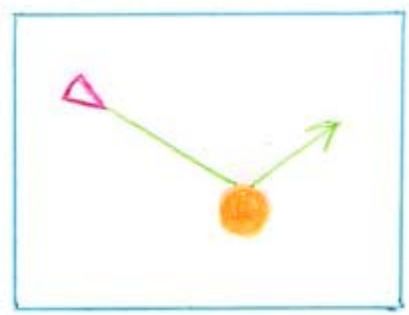
Algorithm : Alignment Rule

1. Alignment_vector = destination of original bird
2. bird1 = original bird
3. bird2 = 2^{nd} bird being compared to original bird1
4. **while** more birds in flock
5. **if** bird2 is within neighbourhood distance to bird1
6. Alignment_vector = Alignment_vector + destination of bird2)
7. **end if**
8. **end while**
9. Alignment_vector = Alignment_vector / number of birds in neighbourhood
10. **return** ((Alignment_vector – destination of bird1) / 8) * weight

4.2.5 Obstacle Avoidance Rule

Birds must avoid hitting obstacles as they fly while still obeying all the other rules of cohesion, separation and alignment. Each time a bird is updated it checks each obstacle in the obstacle array. When an obstacle is found to be within a minimum set distance to a bird, the bird moves away from the object in order to avoid it. This rule has a high weighting as it would be important for real birds not to fly into obstacles and this is reflected in the simulation.

Currently (and in line with most other flock simulations examined) when a bird comes within a set minimum distance to an obstacle it avoids it by almost "bouncing" away. This is by far the easiest method to implement and looks quite natural due to the softening of harsh angles by the presence of surrounding flock-mates, as the other steering rules also continue to affect the birds behaviour thus reducing the angular effect of this movement.



It is achieved by subtracting the co-ordinate position of the obstacle from a vector initially of (0, 0), then subtracting the birds current position from the resultant vector. It remains correct for negative co-ordinate positions as well as positive ones, as subtracting a negative number will effectively add its positive value (e.g. 10 – (–5) = 10 + 5 = 15).

Algorithm : Obstacle Avoidance Rule

1. obstacle_avoid_vector = new vector (0, 0)
2. **while** more obstacles in world (in array of obstacles)
3. **if** bird is within collision distance to obstacle
4. obstacle_avoid_vector = obstacle_avoid_vector – (position of obstacle + half radius of obstacle) – position of bird
5. **end if**
6. **end while**
7. **return** obstacle_avoid vector * weight

4.2.6 Predator Avoidance Rule

The Predator Avoidance algorithm is the same as that used for stationary obstacle avoidance. If the bird is within a preset "predator avoid distance" the avoidance rule will be applied. Again a weight is applied to the rule to increase it's importance and influence. When the Predator is "On" and the bird is within range, the bird obeys only Predator avoidance, Obstacle avoidance and Separation rules – it flies away from the predator while avoiding hitting anything else. Alignment and Separation rules are ignored until the bird is "safe" as it was felt this was more likely to be how real birds would act. When a bird is "caught" by the predator, that bird is "eaten", being permanently removed from the array and from the simulation.

Algorithm : Predator Avoidance Rule

1. predator_avoid_vector = new vector (0, 0)
2. **if** bird is closer than minimum distance to predator
3. predator_avoid_vector = predator_avoid_vector – (position of predator – position of bird)
4. **end if**
5. **return** predator_avoid_vector * weight

4.2.7 Supporting Algorithms:

4.2.7.1 Euclidean Distance:

For all the above steering rules the distance between birds or birds and obstacles needs to be calculated. This is accomplished by calculating the Euclidean distance between them. This is a straightforward calculation using well established mathematical formula.

To calculate Euclidean distance between two points, (x_1, y_1) and (x_2, y_2) in two dimensional space (used in the 2D simulation) the following formula is used [WEI08, WIK08]:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

To calculate Euclidean distance between two points, (x_1, y_1, z_1) and (x_2, y_2, z_2), in three dimensional space (for 3D implementations), the formula is basically the same with the addition of the extra dimension co-ordinate (the z co-ordinate) [WEI08, WIK08]:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

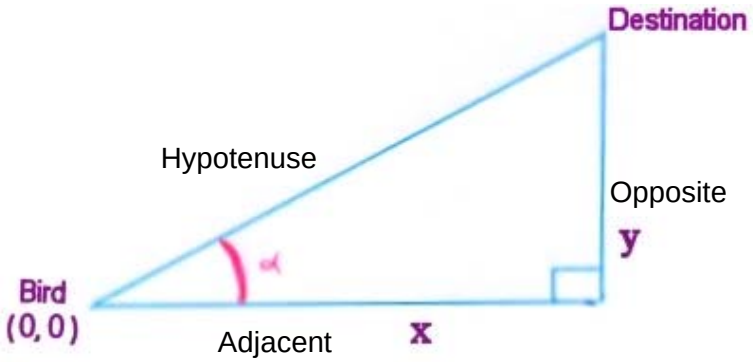
These formula's are very straightforward to implement into code using the position co-ordinates of the birds and obstacles, as the following example in C# shows.

C# Code:

```
public float EuclideanDistance(Bird bird2)
{
      // x = (x2 – x1)^2
      float x = (float)Math.Pow((bird2.BirdPosition.X –
            bird1.BirdPosition.X), 2);

      // y = (y2 – y1)^2
      float y = (float)Math.Pow((bird2.BirdPosition.Y –
            bird1.BirdPosition.Y), 2);

      // returns square root of (x + y)
      return (float)Math.Sqrt(x + y);
}
```

4.2.7.2 Angle Bird Facing:

In order for the bird to be drawn on the screen facing in the direction of travel the rotation angle of the birds destination needs to be calculated. While there are several ways to calculate this mathematically (sine, cosine or tangent) the tangent value was selected as it takes the Adjacent and Opposite sides of a triangle as its arguments, which are effectively the x and y co-ordinates of the birds destination.



Bird is at position (0, 0) on the above diagram. To calculate the tangent of the destination, the destination x co-ordinate (Adjacent) is divided by the destination y co-ordinate (Opposite):

	Tangent = Adjacent / Opposite
Or	Tangent = x co-ordinate / y co-ordinate

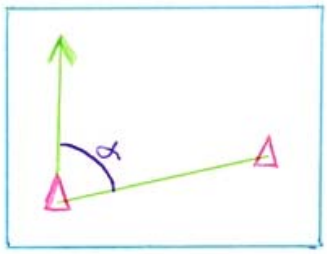
Once the tangent value has been calculated this can be converted either into radians (by a math library function in this particular implementation) or into degrees by standard mathematical tables.

Example:

A bird has (x, y) destination co-ordinates of (300, 500).
Tangent = 300 / 500 = 0.6
Tangent of 0.6 = 0.54105207 Radians [WIK08c]
Which equals 31 degrees [WIK08c]
Therefore the bird is facing 31 degrees from (0, 0)

In this particular implementation (using VS2008 and XNA 3.0) the result is converted to radians using functions from the XNA Math library, then 90^o in radians is subtracted from it to correct rotational offset (otherwise the birds fly sideways). It was also discovered after some experimentation that if the x value of the destination is positive, 180^o in radians must be added to the resultant figure to produce the correct rotation. This points the bird in the correct direction.

4.2.7.3 Angle Between Birds:



This function calculates the angle between the heading of one bird and the position of another. It is used to determine whether a bird is considered a "neighbour" and for predator avoidance. It uses the same formula as above (Angle Bird Facing) to determine the bird's heading, then uses the same algorithm again to determine the relative angle of the second bird's position in relation to the first. The angle of the second bird's position is then subtracted from the angle in which the first bird is facing to calculate the angle between the birds.

In the following algorithm the tangent is converted to radians by a Math library within the XNA framework. This is called, passing in the tangent value and returning the radian value to the calling function. Bird1_destination_X refers to the x co-ordinate of bird1's destination; bird2_position_Y refers to the y co-ordinate of bird2's position. Others are similarly named.

Algorithm: Angle Between Birds

1. tangent_bird1 = bird1_destination_Y / bird1_destination_X
2. radians_bird1 = (*Math function: tangent_bird1 converted to radians*)
3. bird2 _X = bird2_position_X – bird1_position_X
4. bird2 _Y = bird2_position_Y – bird1_position_Y
5. tangent_bird2 = bird2_Y / bird2 _X
6. radians_bird2 = (*Math function: tangent_bird2 converted to radians*)
7. bird1_heading_in_degrees = radians_bird1 * 180 / Pi
8. bird2_position_in_degrees = radians_bird2 * 180 / Pi
9. difference_in_degrees = bird1_heading_in_degrees – bird2_position_in_degrees
10. **return** difference_in_degrees

4.2.7.4 Bird Distance Controller:

This is a vector normalisation function created to control the speed of the birds across the screen and keep their movement reasonably constant as you would expect from real birds. Due to the nature of the vector calculations it is quite possible for the resultant destination of the bird to be co-ordinates of several thousand (either positive or negative in value). The window size is set at 1050 wide, by 700 high (i.e. x = 1050, y = 700) so clearly a destination of say (5000, 6000) is unnecessary and will lead to huge problems with the speed of the bird's movement due to the fact that distance travelled on each update is a set fraction of the total distance to be travelled. If the total distance is huge then the fraction of this distance results in the birds moving too rapidly across

the screen. In order to control this and keep the movement of the birds reasonably even this method reduces the destination co-ordinates to a more reasonable value.

The way it works is quite straightforward. If the destination co-ordinates are greater than twice the width or height of the screen they are simply halved and a small amount added back on to ensure the actual direction is not affected and that the destination is still sufficiently off the screen to keep the birds moving at a reasonable speed. This appears to produce realistic looking results where the birds speed does change very slightly as they move, turn and avoid predators, but in a way which looks quite natural. This does not change the birds' direction, as for example an x co-ordinate of 1,500 or 10,000 are 2 points along the same line.

4.2.7.5 Speed Inhibitors:

As mentioned previously the bird's new position is calculated by multiplying their newly calculated destination by the a hard coded "speed limiter" and then by a user-controlled "speed restrictor" variable. This resultant vector is then added to the bird's original position to give the new position. The speed limiter and speed restrictor were combined to reduce unnecessary code and variables, however this resulted in reduced control over the speed producing disappointing results. They were therefore separated back into two independent variables as results are far better using this method.

The two control mechanisms used to govern the speed of the bird's movement are the speed limiter and the speed restrictor. These are required to produce more realistic results otherwise the birds move so quickly that they appear to simply flash randomly over the screen with no order or pattern to their behaviour, and with such rapidity that the human eye cannot follow any directional movements present.

- The "**speed limiter**" is a hard coded static variable implemented to control the overall speed of movement. It is required so that the bird only moves a tiny fraction of the distance towards it's destination on each update rather than all the way to it's destination, otherwise the bird would not "fly" but rather simply "jump" straight to the new destination on each update. Each time the bird is updated (around 60 times per second), the bird's position changes towards it's destination by the fraction of it's destination set by the speed limiter – a setting of 0.015 has been found to give optimum results. For example if the bird's destination (x, y) co-ordinates were (200, 400), then this would be converted to (200 * 0.015, 400 * 0.015), giving final distance co-ordinates of (3, 6), which would be the actual distance the bird would then travel on this update, moving 3 units along the x co-ordinate from it's current position towards it's ultimate destination and 6 units along the y co-ordinate.

- The "**speed restrictor**" is a user controlled variable which can be increased or decreased within preset boundaries via keyboard input. The higher the number the faster the birds will appear to move, and conversely the lower the number, the slower they will appear to move. A range between 0.05 and 2.0 seems adequate, as any higher than about 0.8 and they move unrealistically fast (up to 2 actually looks quite nice artistically so has been included as the flocking produces quite interesting patterns). Any slower then 0.05 tends to make them look mechanical and jerky rather than flowing gracefully.

Algorithm : Final "Update Position of Bird" Algorithm with Speed and Distance Inhibitors

1. **procedure** NEWPOSITION
2. OldDest = current destination of bird
3. NewDest = updated destination of bird (initially NULL)
4. vectorSum = sum of all vectors returned by behaviour rules functions (cohesion, separation, alignment, obstacle avoidance, predator avoidance)
5. speedLimiter = static variable which limits speed of movement
6. speedRestrictor = user controlled variable to vary speed of movement
7. NewDest = OldDest + vectorSum
8. **if** NewDest > 2 * screen size
9. NewDest is normalised by a reduction function
10. **end if**
11. NewPos = new position of bird
12. OldPos = old position of bird
13. NewPos = OldPos + NewDest * speedLimiter * speedRestrictor
14. **end procedure**

4.2.7.6 Birds Basic Behaviour – Individual Destinations or Follow a Leader:

4.2.7.6.1 Birds with Individual Destinations

It may be observed when watching real birds flying that an individual bird heads in its own direction when it is alone, however if it subsequently joins other birds its destination changes to match that of the flock and it's prior destination is often apparently abandoned. It was felt therefore that for this simulation this was the most realistic approach to follow – each bird has its own individual destination at all times. When it encounters another bird or birds its original destination is altered in accordance with the steering rules applied to it at that time, but always remains individual to that bird. This approach appears to be the method used by other authors of flock

behaviour applications [REY01, GRU07, BUC05, LAL08, RIC07] and certainly gives the best results in this particular simulation. It is therefore the default behaviour for this simulation.

4.2.7.6.2 Birds Following a Leader

The leader / follower algorithm was investigated as part of the project to observe different behavioural outcomes with the leader bird having it's own destination and the follower birds having no destination of their own but rather simply following the leader and adhering to the steering rules. Although this is an unreal situation as far as real birds are concerned it was explored as an option within the simulation to compare the different behaviour generated.

The leader bird has reduced steering behaviours. The cohesion and alignment rules were removed as a leader "leads", so does not want to align itself to the destination of the other birds, nor does it particularly wish to join a flock. The separation and obstacle avoidance rules were originally still applied so that it did not bump into other birds or obstacles. The followers obeyed all the steering rules which are selected but instead of considering their own individual destinations they consider only that of the leader bird. This was found to be unsuccessful as the leader bird was immediately chased off the screen edge by the flock. As it reappeared on the opposite side of the screen (having wrapped around to the other side), all the followers quickly flocked towards it and due to the separation rule it is simply chased off the screen repeatedly.

The separation rule was therefore removed from the leader to try to keep it onscreen for longer, rather than being chased immediately off. While this seemed to improve the overall effect dramatically the follower birds tend to collide a great deal which they do not do when they have individual destinations. Overall it is not successful behaviour and has been left as an option only so the user may explore the resultant behaviour for themselves.

The options of individual destinations or following a leader may be changed dynamically during runtime by the user via the graphical user interface. There are therefore different "update" procedures for the bird objects depending upon which option is chosen at the time.

A design decision was made not to make a separate class for the leader bird as the option is turned on and off during runtime. Currently a pre-existing bird becomes the leader when this option is selected and rejoins the flock when the option is turned off. While it would have been entirely possible to write a separate class it was felt that better performance would be achieved by using an existing bird.

4.2.7.7 Random Destination Change for All Birds:

In nature it may be observed that a flock of birds may suddenly alter course for no apparent reason so this option was created to emulate that behaviour. If the option is selected this function carries out the randomised timing calling the "new destination" function on all birds when appropriate.

It is a very simple algorithm. Two random numbers are chosen between zero and three hundred, and if they are the same then every bird is given a new randomly chosen destination. A range between zero to 300 was chosen as a lesser value results in destination changes being too frequent, while much greater than this and they are not frequent enough.

Algorithm Randomly Timed Destination Change All Birds:

1. numberOne = random number between 0 - 300
2. numberTwo = random number between 0 - 300
3. **if** (numOne == numTwo)
4. call NewDestination function on every bird.
5. **end if**

4.2.7.8 New Destination for Birds:

When birds move off the screen edge they are wrapped to the opposite side and given a new randomly chosen destination. To provide a new destination the Random_Number_Generator is called for both the x and y co-ordinates (one at a time), passing in as the limit the screen width and height respectively, resulting in return of a number between zero and the value passed in. As the Random_Number_Generator always returns a positive number it is called a second time for each of the x and y co-ordinates to return a value of either zero or one (effectively flipping a coin). If a zero is returned the value for that co-ordinate remains positive, if a one is returned the co-ordinate value becomes negative. This provides the bird with a new destination which may be either positive or negative co-ordinates within a specified range.

5: DESIGN AND DEVELOPMENT:

5.1 Implementation Platform And Language

Microsoft Visual Studio and the .Net framework was chosen as a development environment implementing C# as the chosen language due to some degree of existing familiarity with both and the ready availability of support documentation and tutorials. Visual Studio 2008 also has fully integrated support for test suites and is reputedly the "best of breed" in this type of application being widely used in industry [MAN08a].

Having chosen the environment, framework and language, a Software Development Kit was required providing a collection of pre-built programming components such as libraries for carrying out many of the "gaming" functions such as drawing sprites and backgrounds etc. on the screen and providing an interface for user controls [MIL08]. Several were considered although choice was limited somewhat as most are language specific with C# appearing reasonably unpopular in general. The Microsoft XNA framework was finally chosen for several reasons: it integrates seamlessly with Visual Studio (having been designed specifically for this purpose), it is specific to the C# language, it has a plethora of online tutorials and support available both from Microsoft and independent sources, and it is currently seen as "cutting edge" software within the games development industry [MAN08]. It also provides a very efficient inbuilt Model-View-Controller design pattern for easy implementation which obviates the need for direct use of delegates and interfaces by the programmer.

The project was initially started using Visual Studio 2005 with XNA Framework 2.0 and while this appeared to provide adequate performance in most respects Visual Studio 2005 does not provide support for test suites. XNA 2.0 is not compatible with Visual Studio 2008 but is still the only official full release version of XNA available which is why it was chosen initially. Due to the lack of testing abilities using this combination however it was decided to upgrade to Visual Studio 2008 and use the XNA Framework 3.0 CTP (Community Technology Preview) version to allow the integration of testing within the project, although this beta version of XNA 3.0 lacks several features which would have been desirable including the ability to publish the project to a standalone application.

5.2 Feasibility:

Resources required were realistic and readily available. The only requirements were a computer capable of a reasonable level of graphics and processor performance, with the chosen software / development platform installed, and the required software being readily available. Tutorials, books

and information on game programming in general and Visual Studio, C# and the XNA framework in particular appeared plentiful. As there was a time restraint on the project a timetable was drawn up to ensure that work progressed at such a rate so as to complete the project within the set time.

5.3 Object Oriented Approach and Class Organisation:

An object oriented approach was chosen as this was seen as suitable for this type of project with obvious "objects" such as birds and obstacles required.

Calculations where there is a possibility of dividing by zero have preventative checks put in place with co-ordinates of 0 being changed to 0.1. All such calculations use float values. While this would not be acceptable in many critical systems where accuracy is essential, in the instance of this simulation it was felt this would not have any detrimental effect on the birds' overall behaviour, being minute changes imperceptible to the human eye. It prevents division-by-zero exceptions being thrown.

Full code for the simulation is presented in Appendix D.

5.3.1 Overview of XNA Provided Skeleton:

When a new project is created using XNA Game Studio 3.0 a basic programme skeleton with classes and empty methods is created upon which to build consisting of a **Program class** containing only a Main method which creates a new simulation (game) and starts it running, and a **Game1 class** which then takes over and controls the overall running of the simulation (see Appendix F for XNA provided skeleton code).

5.3.2 Class Organisation:

5.3.2.1 Game1 Class:

The Game1 class contains methods for loading the initial content of the simulation, regularly updating the objects and member variables, and drawing the graphical representation on the screen. It contains functions such as a Constructor (which sets up control of the graphics device and the simulation window parameters, and allows recognition and retrieval of button clicks, keystrokes etc. for user input), Initialize (which performs initialization of content required prior to the simulation running such as member variables, creation and population of arrays, and setting up the window in which the simulation runs) and LoadContent (which sets up sprite batches and sprite

fonts and loads content such as textures and positions into objects such as birds, obstacles, and fonts).

Once the simulation parameters are set up and the simulation starts running the whole simulation is basically controlled by the Update function in this Game1 class and represented graphically using the Draw function.

The Update function is called 60 times per second and allows the game to gather input and update the world. It calls methods in the Bird classes which update positions and destinations of all FlockBirds and the Predator. Update also calls other functions within the Game1 class which "listen" for user input such as keyboard strokes or mouse clicks and when these are present the appropriate actions are carried out such as turning steering and behaviour rules on or off, altering the speed of the birds movement, placing obstacles on the screen where indicated by the user, or creating a user selected destination for all birds. It also calls the function which updates strings displayed on the graphical representation such as "On" or "Off" for behaviour rules and one which controls random destination changes for the birds if this option is selected. All functions which require updating are called by this Update function and thus it is responsible for the overall running of the entire simulation.

The Draw function is also in the Game1 class and this is responsible for drawing a representation of the simulation on the screen at regular intervals including the actual window, the birds and obstacles, and the user instructions and current status of all steering and behaviour rules. Also displayed in real time are the vectors for position and destination of up to 5 birds and the number of birds currently in the flock array. The actual Bird, Obstacle and UserTarget objects are drawn by their own separate Draw functions within their respective classes which are themselves called by the Game1 Draw method.

5.3.2.2 Bird Class:

This is a base class with inherited classes of FlockBird and Predator. It contains member variables which are used by both types of bird such as position and destination of the bird, size of world, obstacle avoidance distance, texture sizes and speed limiters.

Many of the methods and functions in the Bird class were discussed in detail above (Chapter 4), and were placed in the Bird base class as they are used by both FlockBirds and Predators, both of which inherit from the base class. These include:

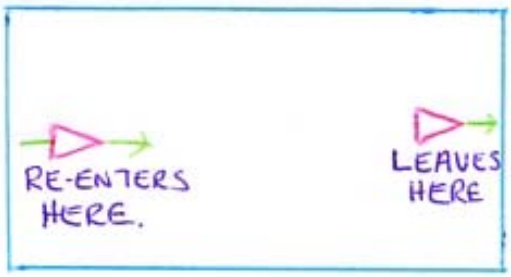
- Cohesion rule
- Obstacle Avoidance

- Euclidean Distance which includes 3 versions:
    - To calculate the distance between 2 birds.
    - To calculate distance between a bird and an obstacle.
    - To calculate distance between a bird and a set of vector co-ordinates.
- Angle Bird Facing
- Angle Between Birds
- Bird Distance Controller

Like the Game1 class, the Bird class also has its own LoadContent function which loads the size of the bird "texture" as well as the minimum and maximum co-ordinates within which the birds may travel (i.e. size of the world). It also has its own Draw function which draws the bird sprites on screen, and this is called by the Draw function in the Game1 class.

There are also functions which place new birds within a specified area on the screen, provide birds with new randomly selected destinations (using the RandomNumberGenerator class), and "wrap" the birds to the opposite edge of the screen when they move outside the boundaries of the world – as if the 2D world is a globe opened out.

New birds placed in one area on screen



Birds "wrapped" to other side when they fly off the screen.

In order to draw the bird facing in the direction in which it is travelling the Draw function calls the Angle_Bird_Facing function (4.2.7.2) which calculates the rotational angle of the bird and returns the value in radians to the Draw function so the birds may be drawn correctly on screen.

5.3.2.2.1 FlockBird Inherited Class:

As an inherited class the FlockBird accesses all the functions of the Bird class as well as having it's own functions and member variables. Almost all of the FlockBirds functions are discussed in detail in Chapter 4 including:

- Separation Rule
- Alignment Rule
- Predator Avoidance Rule

- Calculate Corrected Heading for all Birds (Birds Basic Behaviour – Individual Destinations or Follow a Leader)
- Update Bird (Update Bird's Position and Destination)
    - Individual
    - Leader
    - Follower

The only other main function apart from the constructor within the class is the User Set Destination function which simply sets the bird's destination to the point on screen where the user has clicked.

5.3.2.2.2 Predator Inherited Class:

Besides the Constructor which calls the base class Bird constructor the Predator class contains only two methods. Calculate_New_Heading calculates the new destination for the predator using only the Obstacle Avoidance rule and the Cohesion rule. The cohesion rule is the means by which the predator chases the flock birds, by trying to flock closely with them. This is called by the Update_Predator function which checks that the predator is still within the world area (if not, it wraps it around the world and provides it with a new destination) and calculates the predator's new position taking into account it's current position, it's new destination (calculated from the steering rules), vector normalisation (Distance Controller) and speed limiters in the same way as the FlockBirds are given new destinations and positions.

The predator has the ability to move faster than flock birds so that it may "swoop" on its prey in a rapid motion. If it comes within a set distance of a FlockBird it "eats" it by calling the EatBird function in the Game1 class which removes the bird from the Array List and redefines the flock array.

5.3.2.3 Obstacle Class:

Like the other classes the Obstacle class has a Load Content function (which loads the texture for the obstacle, the height and width of the texture, and the co-ordinates of the centre of the texture), and a Draw method (which draws the obstacle sprites on screen).

Another function moves the two original hard coded obstacles to the desired co-ordinates. When the Obstacle Array is created Obstacles are created with (0, 0) co-ordinates, before being moved into their positions by the Game1 Load Content method calling the PositionObstacle method within the Obstacle class.

It was decided to limit the number of obstacles to ten otherwise the screen becomes too crowded and the birds cannot “fly” but instead simply shuffle between obstacles. Two obstacles are hard coded into place and the user has the option of placing a further eight where-ever they wish.

5.3.2.4 User Target Class:

The User Target class was actually created as an inherited class of the Microsoft.Xna.Framework.DrawableGameComponent class. As such it contains the same basic empty functions as the Game1 class. The constructor and the “Initialize” function simply call the base class. The “Load Content” sets up a Sprite batch and loads the picture (texture) into the object so it can be drawn on screen.

When the target sprite is drawn on the screen it remains visible for approximately two seconds before fading away. This is done by decrementing a counter. While the counter is above zero the target is visible but incrementally fading. Once it reaches zero it has faded completely. The “Draw” function draws the target texture on the screen in the appropriate position, and like the Draw functions for the Bird and Obstacle classes it is called from the Draw function in the Game1 class.

5.3.2.5 Random Number Generator:

Another very small class, the Random Number Generator takes as an input parameter a positive integer number and returns a randomly generated number between zero and the integer passed in. It uses an inbuilt library function to do this. It was placed into a separate class so that all other classes could access it freely simply by passing in the maximum number of the range required.

5.3.3 Encapsulation

Class member variables were encapsulated providing a public interface to them while maintaining a private implementation which provides an increased level of security and a means of validating values before assigning them to the member variables. Although this simulation is a standalone application rather than a component in a larger project, it was felt that good practice should be maintained at all times and as encapsulation is generally viewed as good practice it was implemented into this simulation.

As the example below shows, the private variable speedRestricter has been encapsulated allowing public access to "get" the value and also to "set" a new value for the variable. Setting the new value has restrictions placed upon it to ensure that only values within a certain range are accepted. This protects the variable from invalid values while allowing users outside the class to alter it.

All variables which require access from outside their own classes have been encapsulated for this security control.

Example of Encapsulation:

```
private float speedRestricter = 0.2f;
//Controls the speed of the birds
```

Above variable is "private". Minimum and maximum values may be set for public access to the private member variable. This example ensures that the speedRestricter is limited between 0.05 and 2.0.

```
public float SpeedRestricter
{
      get { return speedRestricter; }
      set
      {
            speedRestricter = value;
            if (speedRestricter <= 0)
                  speedRestricter = 0.05f;
            if (speedRestricter > 2)
                  speedRestricter = 2f;
      }
}
```

5.4 Agile Approach and Iterative Development:

An Agile Development Approach was used which meant an iterative approach was implemented starting with the smallest possible viable program which was basically just a single bird flying around the screen with no rules to follow, then gradually building upon it adding more functionality, behavioural rules and features on each successive iteration. This was seen as a superior approach to the waterfall method as it allowed for gradual development, improvement and refinement of the program and was compatible with test driven development.

5.4.1 Iterations Implemented during Development (Brief overview):

First iteration:

Development of a basic framework of the program, producing the graphical environment in which a single sprite (bird) flew around the screen with a hard-coded destination. This was a huge leap from any prior programming experience I had on the course as it involved game programming and graphics, neither of which I had any familiarity.

Second iteration:

Removal of the hard-coded destination making it randomised or arbitrary. An array was also added to create a "flock" of birds. At this stage all birds shared a common destination. A user option was added whereby the user could click somewhere on the screen and the destination of all birds would be set to that point. A test project was also added to support test-driven development and the environment was switched to VS2008 with XNA 3.0. The Bird class was created moving all the code for the Bird objects out of the main Game1 class into a separate class of its own

Third iteration:

Introduction of the flocking behaviour rules by which the bird objects interact with each other, moving together as a flock rather than simply randomly on screen. This was obviously the main focus of the project and therefore one of the biggest iterations, being itself broken down into three mini-iterations with one behavioural rule being added at a time. These were tested separately as they were introduced, and together as further rules were added, checking that they worked correctly giving the expected results. Fine tuning was frequent and extensive at this stage to ensure the behaviour rules worked in the intended manner.

Fourth iteration:

An option was added so that the birds could either fly with individual destinations as before, or could be made to follow a leader bird. This required quite different behaviour from both the leader bird and the followers.

Fifth iteration:

Hard coded obstacles were introduced with an additional behavioural rule of “obstacle avoidance”.

Sixth iteration:

A user interface was added providing the user with options of turning all steering and behaviour rules on and off, as well as controlling speed of the birds, whether they follow a leader or have individual destinations, and allowing randomly timed destination changes on all birds.

Seventh iteration:

User placement of obstacles was added. Steering rules were updated to work correctly as they were not producing the required behaviour. Weights were added to rules to allow their influence levels to be altered. A Pause option was added to the interface.

Eighth iteration:

Bird class was refactored into base class and inherited classes of flock birds and predator bird. Neighbourhood of bird was refined so that it only considers other birds which are within it’s “field of vision” (i.e. ignoring birds behind it), although this was not successful at this stage. Predator bird was added and predator avoidance rule added for flock birds to flee from the predator.

Ninth iteration:

Bird Distance Controller function was discovered to be producing erroneous results and was corrected, resulting in neighbourhood distance now working correctly, as well as many other problematic areas such as predator and obstacle avoidance. The Flockbird array was changed from being a static size to being set up for dynamic alterations in array size so that bird numbers may be increased or reduced. Number of birds now displayed in real time on screen.

5.5 Test Driven Development:

A test driven development approach was chosen for this project as this approach provides several benefits including:

- Provision of continual feedback on code to ensure it is working correctly
- Allows the code to evolve in such a way that it does exactly as it should and no more
- Provides a collection of tests which can be run frequently to ensure that changes to one part of code have not adversely impacted upon another part of the code. [GOR05].

Visual Studio 2008 provides inbuilt support for test suites, both writing the test class and running the tests within the environment, thereby keeping the test code with the source code in the same project at all times for easy testing and integration. Some things were unable to be tested such as the actual visualisations on the Graphical User Interface (e.g. the Draw methods – a test method cannot "see" whether the birds are shown onscreen correctly) – these were assessed visually by running the programme and viewing them directly.

Tests carried out were both progressive (testing new code to check for correct functioning or errors), and regressive (repeatedly retesting old code as new code was added to ensure that new code had not introduced errors into existing code).

The importance of Test Driven Development was proven during this project. Functions were tested as they were developed to ensure correct results however one, the Bird Distance Controller responsible for normalising vectors, was unknowingly omitted from testing. When travelling towards negative value co-ordinates the birds sometimes showed anomalous behaviour acting in a very different manner to that displayed when moving towards positive co-ordinates and sometimes seeming to "spin" as they moved. This continued for some time with the cause remaining unidentified. Upon realisation that a test method for this function was lacking and remedial action being taken it was immediately discovered that the method was producing flawed results, converting negative value co-ordinates to positive values. This obviously had a huge effect on the bird's behaviour and was the cause of this "spinning". The method was corrected with a noticeable improvement in performance of the simulation with the birds' movement becoming much more fluid and graceful, and no longer displaying anomalous behaviour. Without unit testing this error may possibly have remained undiscovered.

Following is the original flawed function, the test which identified the flaw, and the code corrected as a result of the testing.

BirdDistanceContoller function prior to correction of error (Producing flawed results):

```
public void BirdDistanceController(Game1 theGame)
{
   float extraBit = 200;
   if (this.BirdDestination.X > (2 * theGame.DisplayWidth))
       this.birdDestination.X = ((BirdDestination.X / 2) + extraBit);
   if (this.BirdDestination.X < (2 * theGame.DisplayWidth) * -1)
       this.birdDestination.X = ((BirdDestination.X / 2) + extraBit) * -1;
   if (this.BirdDestination.Y > (2 * theGame.DisplayHeight))
       this.birdDestination.Y = ((BirdDestination.Y / 2) + extraBit);
   if (this.BirdDestination.Y < (2 * theGame.DisplayHeight * -1))
       this.birdDestination.Y = ((BirdDestination.Y / 2) + extraBit) * -1;
}
```

Above is the original flawed code for the BirdDistanceController function. Below is the test method which was instrumental in identifying the error in the code, followed by the amended code which now produces correct results. Accurate expected results were calculated manually for the test using birds with set co-ordinates. These birds were then created in the test suite and used to test the function. Results were returned from the function and compared to those calculated.

Test Method for the function:

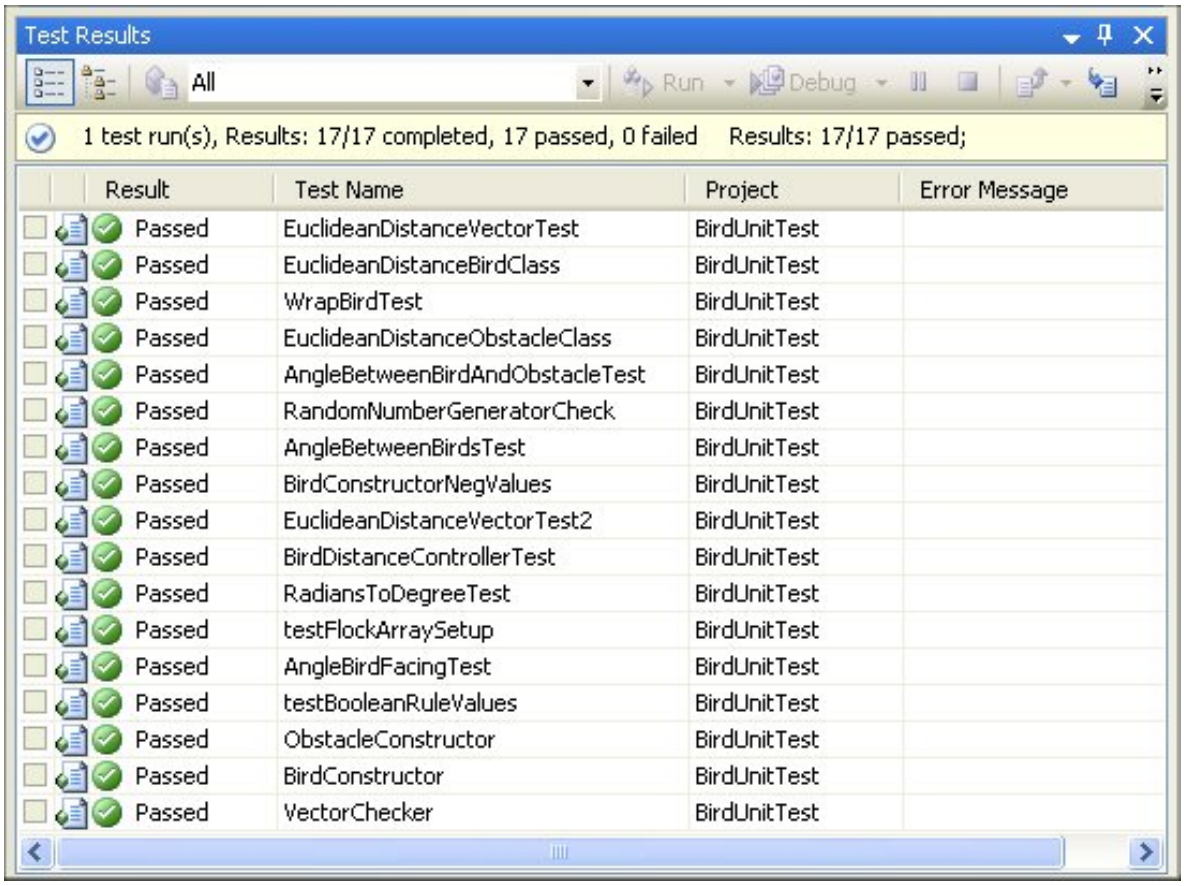
```
        Bird bird11 = new Bird(300, 400, 6000, 3000);
        Bird bird12 = new Bird(200, 300, -4000, -4398);

        [TestMethod]
        public void BirdDistanceControllerTest()
        {
            bird11.BirdDistanceController(FlockingBirds);
            Assert.AreEqual(3200, bird11.BirdDestination.X);
            Assert.AreEqual(1700, bird11.BirdDestination.Y);

            bird12.BirdDistanceController(FlockingBirds);
            Assert.AreEqual(-2200, bird12.BirdDestination.X);
            Assert.AreEqual(-2399, bird12.BirdDestination.Y);
        }
```

BirdDistanceContoller function after Refactoring. Now producing correct results:

```
public void BirdDistanceController(Game1 theGame)
{
   float extraBit = 200;

   if (this.BirdDestination.X > (2 * theGame.DisplayWidth))
       this.birdDestination.X = ((BirdDestination.X / 2) + extraBit);
   if (this.BirdDestination.X < (2 * theGame.DisplayWidth) * -1)
       this.birdDestination.X = ((BirdDestination.X / 2) - extraBit);
   if (this.BirdDestination.Y > (2 * theGame.DisplayHeight))
       this.birdDestination.Y = ((BirdDestination.Y / 2) + extraBit);
   if (this.BirdDestination.Y < (2 * theGame.DisplayHeight * -1))
       this.birdDestination.Y = ((BirdDestination.Y / 2) - extraBit);
}
```

Tests were designed for all functions which could be tested with final results showing correct functioning of all methods. Some methods could not be tested such as the Draw methods, as a test cannot discern whether or not a sprite is visible on screen. Such methods were tested manually by direct visualisation of the screen, placing a sprite at set co-ordinates and measuring it's apparent position with a ruler then comparing this to the total window size. While obviously not precise the results were adequate to show that sprites were being drawn in the correct position and facing the correct direction.

The following diagram shows all tests have passed without errors



Expected results for each test were calculated manually prior to tests being run, with methods being checked and corrected if tests failed thereby ensuring correct functioning.

Code for the entire test suite is provided in Appendix E.

5.6 Refactoring:

Refactoring is making improvements to the existing code without changing it's actual functionality.

Once the code was working effectively it was refactored to achieve lower coupling and higher cohesion and to make it easier to understand and maintain. Between each change all unit tests were re-run to ensure code was still functioning correctly and the simulation was run to check visually whether it was still working as it should.

Initially all methods and functions were in the basic Game1 class as this is how the online tutorials for XNA gaming in general are set out, so as I was learning to use the XNA framework with these tutorials I also used this method to get started. Once a working program had been established however, the code was refactored creating separate classes for Birds and Obstacles so that "objects" of these types could be created and manipulated using an object oriented approach. Separate classes for the User-selected-destination and for the Random Number Generator were also added to increase cohesion and reduce coupling, and so they could be used freely by all other classes.

Functions and methods were also refactored in cases where they were fully functioning but with less than ideal code. "Magic numbers" were removed for clarity and improved maintainability, and code was rewritten to maximise performance by reducing operations which might slow the system down by unnecessarily repeating actions. Although the code is heavily documented for clarity and easy maintenance, self-documenting code further enhances maintainability and was therefore aspired towards, with code being refactored to be self-explanatory wherever possible.

Following the separation and creation of the Bird and Obstacle classes from the Game1 class, there was only one Bird class, therefore when a predator was added to the simulation the Bird class was again refactored into a base class of Bird with inherited classes of FlockBird and PredatorBird. This was to maximise code re-use and minimise repeated code. Functions and code common to both flock birds and predator birds were left in the base class for use with both types of bird, while only code specific to each subclass was moved out into the inherited classes. The majority of member variables remain in the base Bird class as most are common to both Flock birds and the Predator such as position and destination co-ordinates, size of the bird texture, "edge of the world" co-ordinates and speed limiter. The Collision Distance and Predator Avoidance Distance were put into the Flock Bird class as the predator does not use either of these, but rather actively seeks to "collide" with the flock birds in order to eat them. To illustrate this the Class Diagrams for the Bird class(es) before and after refactoring are presented in Appendix G.

Another smaller example of refactoring is presented on the following page. This is the code for the Random Destination Change for all Birds – one of the user options which gives all flock birds a new destination at randomly chosen intervals.

Originally the code was simply an **if** statement within the Update function in the Game1 class rather than being a separate function itself. It used the time played (time the simulation had been running) and two hard coded integers (their actual value being irrelevant). If the time played modulus the first integer was equal to the second integer then all birds received a new destination. It was soon realised however that this was not actually random but produced regular cyclic timings.

Original in Update function as if statement:

```
//numOne and numTwo are hard coded variables
if (randomlyTimedDestChange)        //if the user option is selected
{
      if (timePlayed % numOne == numTwo)
          foreach (FlockBird bird in flockArray)
              bird.NewDestination(this);
}
```

This was refactored out of the Update function into a separate function of its own which is called by the Update function if the user selects the option on the User Interface. The integer numOne was changed to be randomly chosen each time the function was run to avoid the regular cyclic timing as was previously the case and introduce true randomisation.

Code refactored into separate function and randomisation improved:

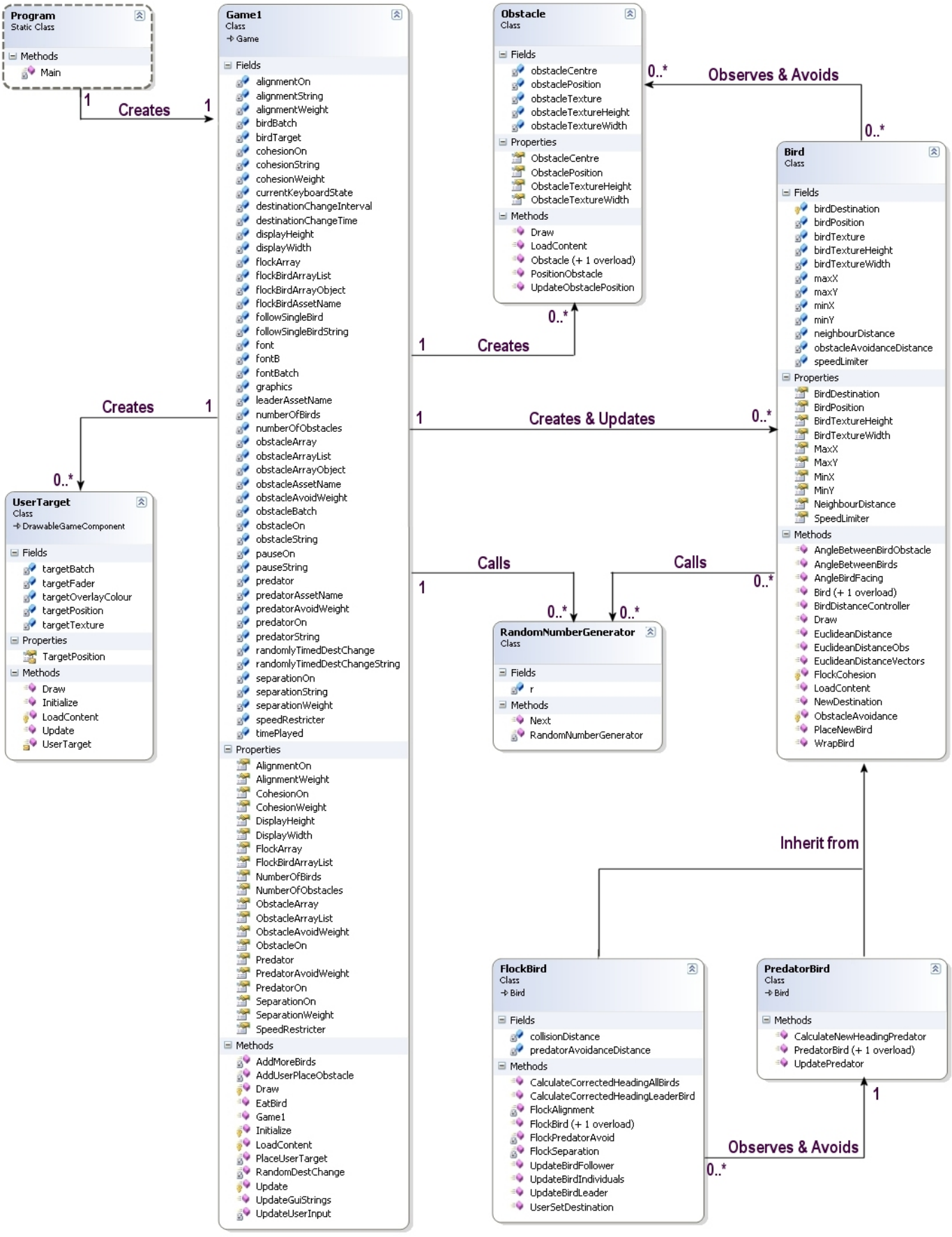
```
private void RandomDestChange()
{
      int numOne = (int)RandomNumberGenerator.Next(200);
      if (timePlayed % numOne == numTwo)
          foreach (FlockBird bird in flockArray)
              bird.NewDestination(this);
}
```

Finally it was decided to improve further upon this by simply chosing two random numbers between zero and 300 each time and if they were the same then birds were given a new destination. If they were different no action was taken. This was seen to be truly random and is the final version of this function. Three hundred was chosen after experimentation as it appeared to provide destination changes which are not too frequent nor too infrequent.

Code refactored again to obtain true randomisation:

```
private void RandomDestChange()
{
      int numOne = (int)RandomNumberGenerator.Next(300);
      int numTwo = (int)RandomNumberGenerator.Next(300);

      if (numOne == numTwo)
            foreach (FlockBird bird in flockArray)
                  bird.NewDestination(this);
}
```

5.7 Class Diagram:

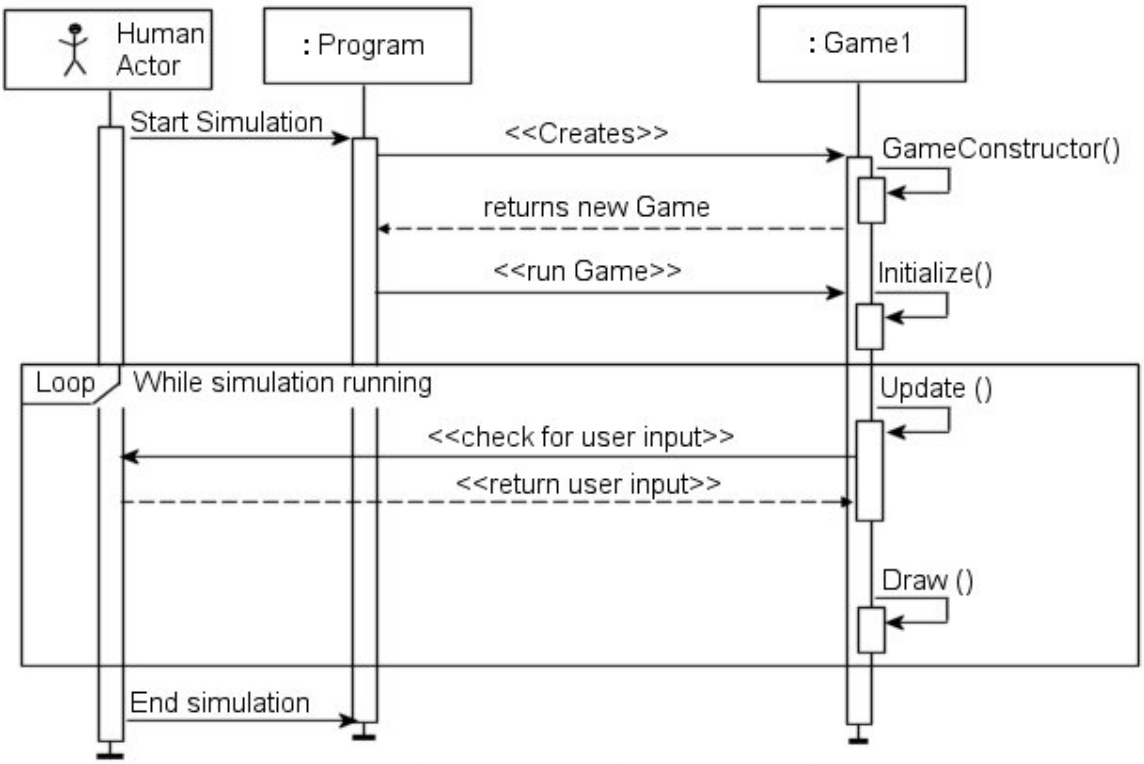


5.8 Sequence Diagrams:

Sequence diagrams have been created for the major functions of the simulation where classes and/or objects interact. Functions which simply carry out mathematical calculations or update variables have been omitted as these are not design issues so much as they are specific coding issues, although diagrams of the Cohesion and Obstacle Avoidance rules have been included as examples.

5.8.1 Overall Simulation:

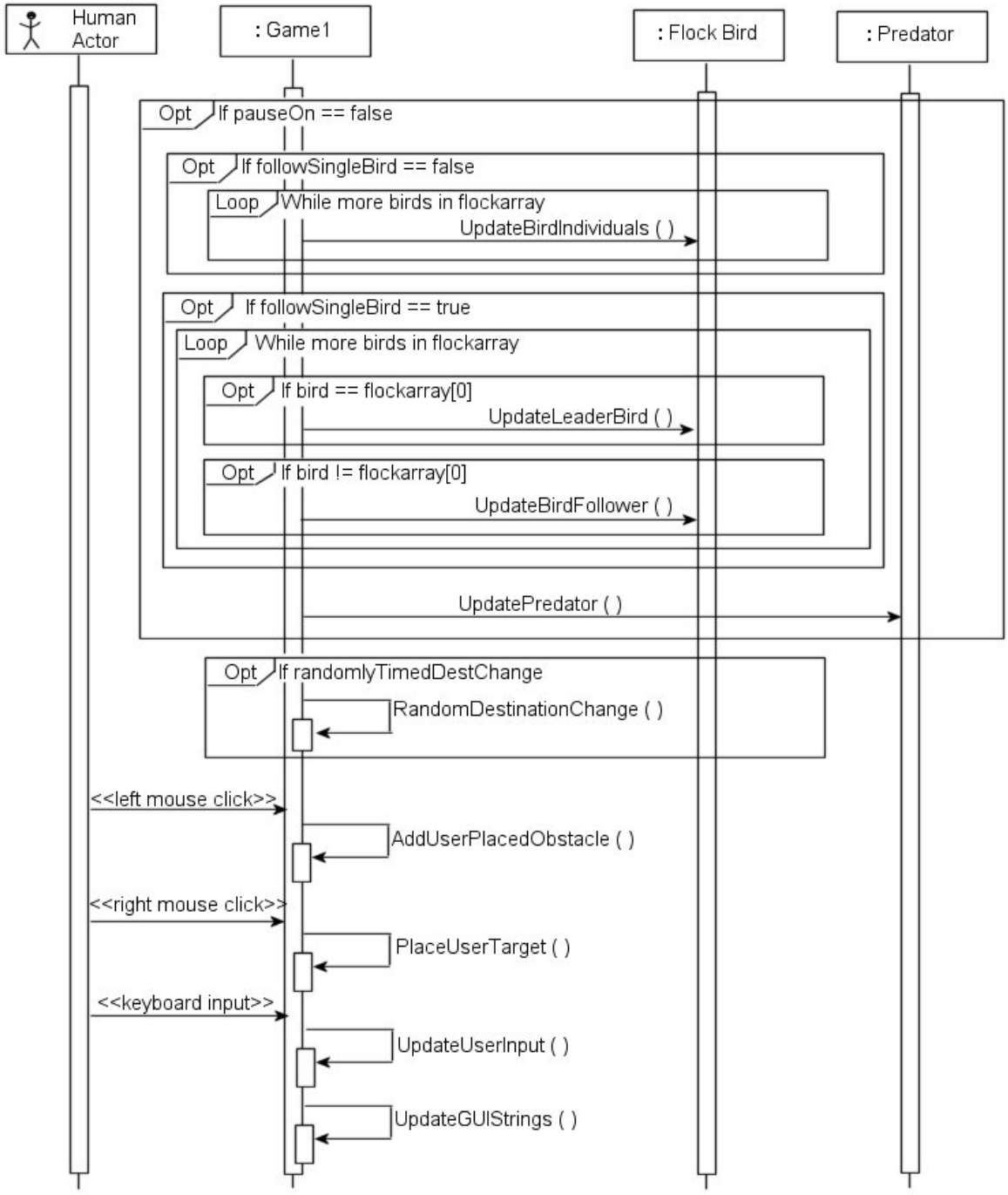
Shows the overall structure of the simulation with the user effecting the creation of a new instance of the simulation, then the simulation running until the user ends the program. The Update and Draw functions are shown as separate diagrams in more detail.



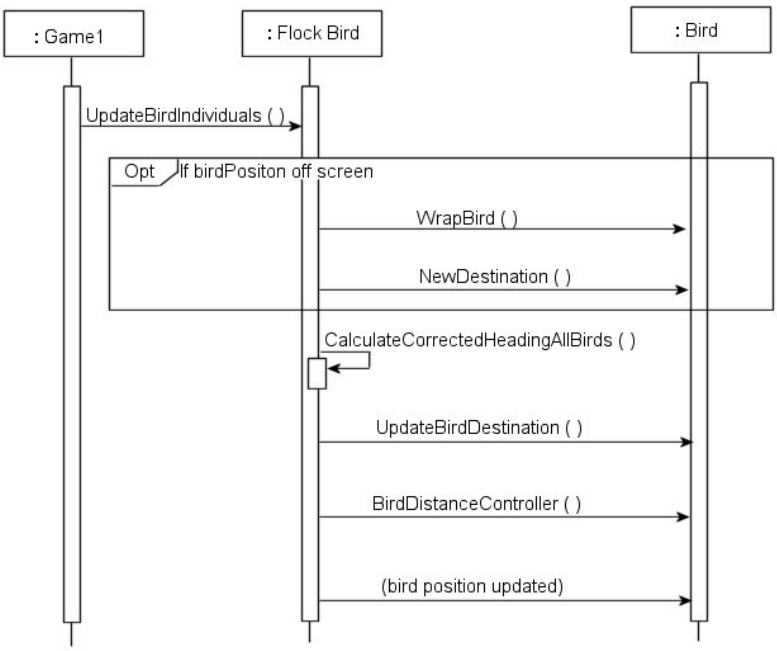
Comment: The Update function runs other separate functions which each check for user input, updating parameters if input is recieved. Refer to sequence diagram for Update function.

5.8.2 Update Function (in Game1 class):

The Update function is called 60 times per second updating many aspects of the simulation. Again, finer granularity diagrams showing specific functions are shown separately.

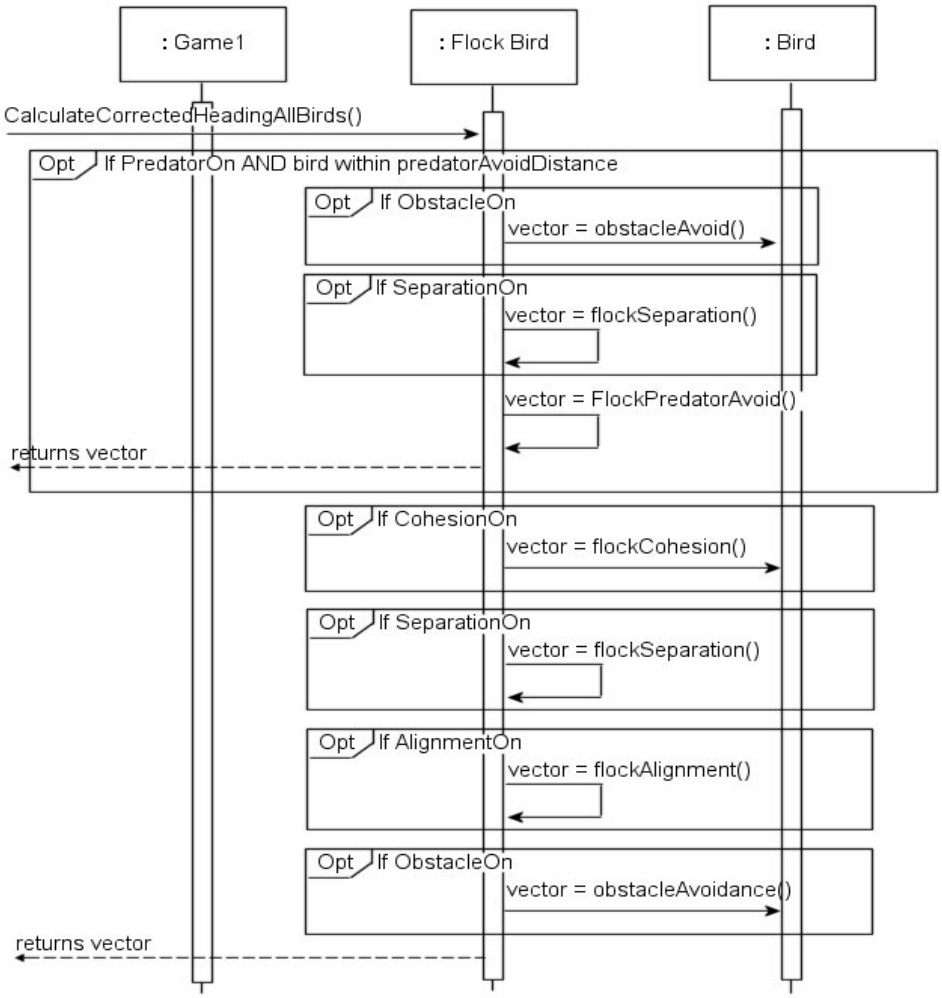


5.8.3 Update Bird Individuals:

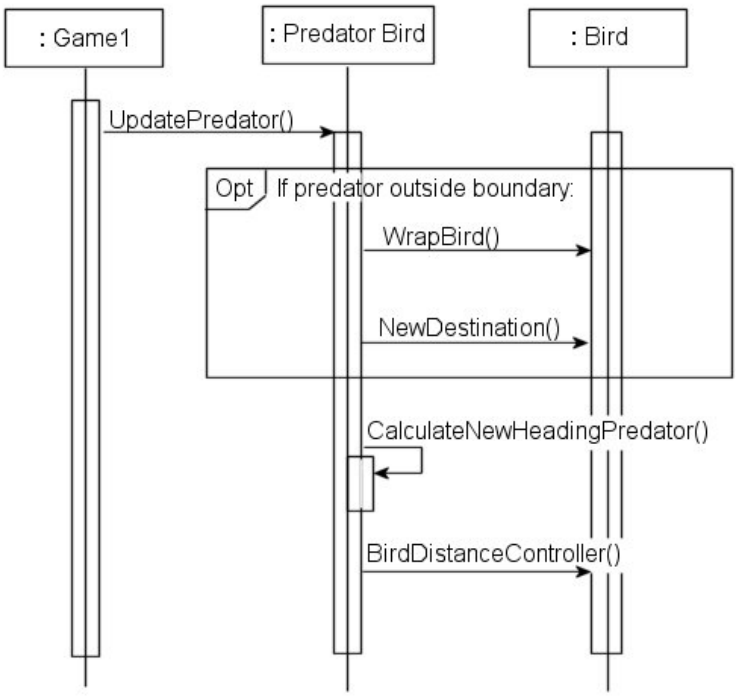


If birds each have their own individual destinations the Update function (previous page) calls **Update Bird Individuals** (right) which itself calls **Calculate Corrected Heading All Birds** (below). Similar functions are called if birds are following a leader bird. These are not shown due to their similarity.

5.8.4 Calculate Corrected Heading All Birds:

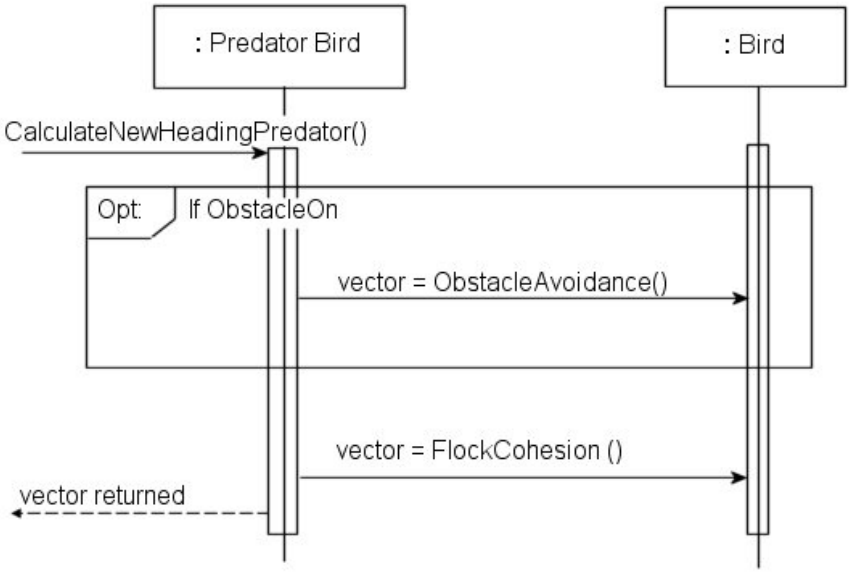


5.8.5 Update Predator Function:



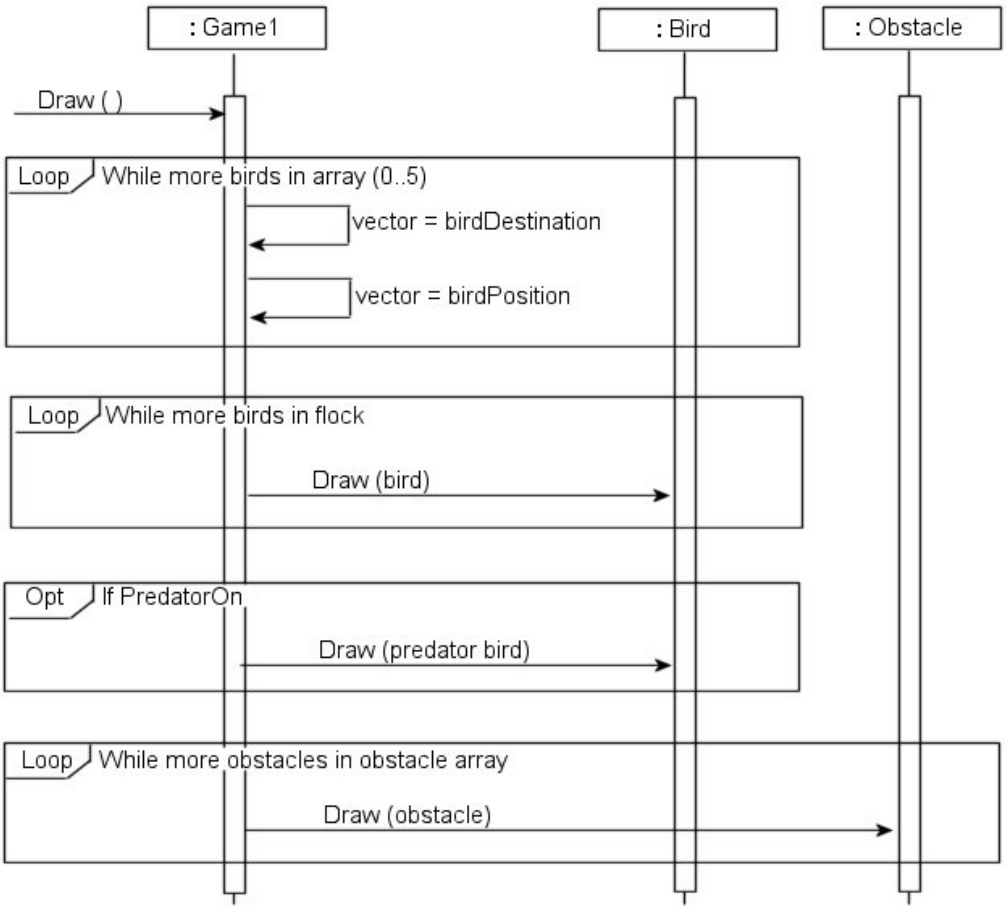
The Update function also calls the **Update Predator Function** which like the flock-bird update above, calls its own **Calculate New Heading Predator** function (below).

5.8.6 Calculate New Heading Predator:

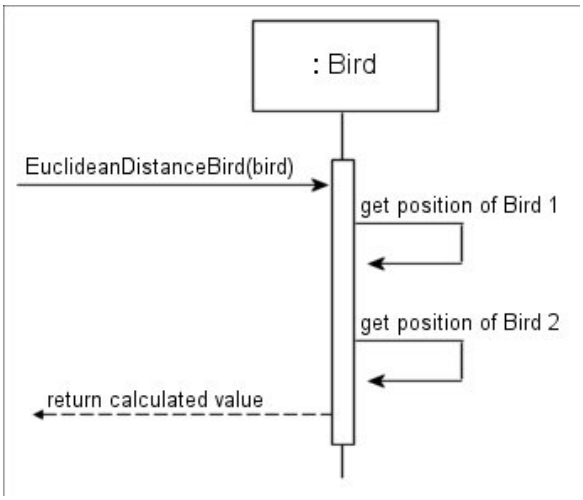


5.8.7 Draw Function (in Game1 class):

The Draw function draws the main window with the GUI elements and calls separate Draw functions in the Bird and Obstacle classes which each draw their own sprites.



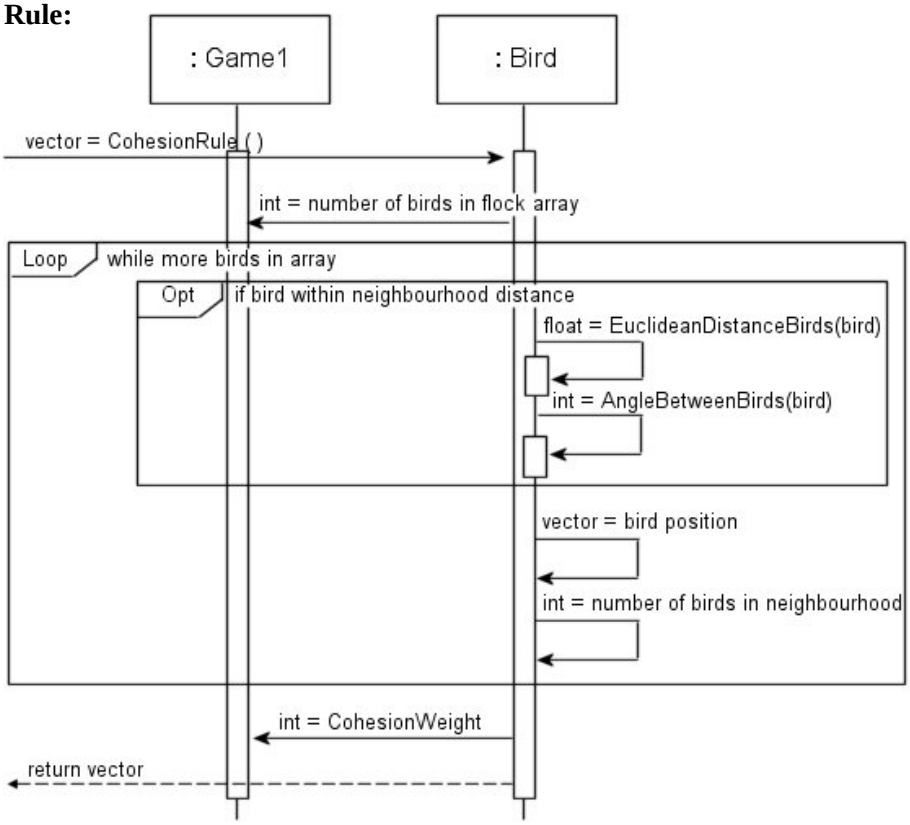
5.8.8 Euclidean Distance:



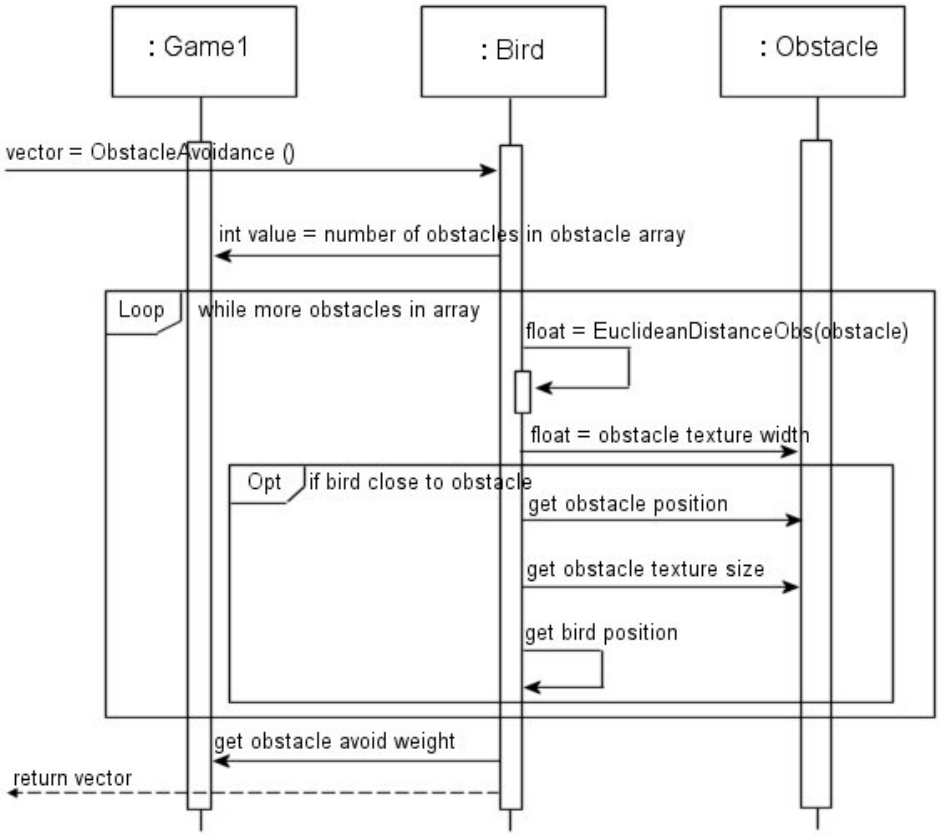
Euclidean Distance is used by the Bird class and its inherited classes. It is a fine granularity function which carries out mathematical functions, not interacting with other classes. The details of its functionality have been left to the specific code.

The Cohesion and Obstacle Avoidance Rules are shown here as two examples of steering rules. It was not deemed necessary to demonstrate all of them, as all are similar in design.

5.8.9 Cohesion Rule:



5.8.10 Obstacle Avoidance Rule:



6: GRAPHICAL USER INTERFACE (GUI):

Numerous user controls have been implemented in the GUI to provide the user with control over the bird's behaviour. The status of all user options is displayed on screen so the user may see easily which options are selected at any time. All are dynamic and may be switched freely during runtime. Brief prompts are displayed on screen to guide user in key selection, and a Quick Guide to User Control Keys is presented in Appendix A.

User controlled options include:

- Turning individual steering rules on and off so that birds behaviour with different combinations of rules may be observed. Controllable rules are: cohesion, separation, alignment, obstacle avoidance and predator avoidance.
- User placement of obstacles on the screen
- Control over speed of the birds movement
- User can click onscreen setting that point as the destination for all birds
- User can change whether all birds have their own individual destinations or whether they follow a leader bird.
- Randomly timed destination change for all birds may be enabled / disabled.
- Predator may be enabled / disabled.
- Birds may be added to the flock up to a maximum of 60 birds (a greater number leads to decreased performance on some machines)

6.1 Steering and Behaviour Rules:

Steering rules may be turned on and off freely during runtime so that behaviour may be observed with various combinations of rules. These are cohesion, separation, alignment and obstacle avoidance. To select / deselect steering rules and other options press the corresponding keys as indicated on screen (keys in bold), and their status will be shown.

6.2 Pause Simulation:

The simulation may be paused (or "held) using the **H** key. This will pause the simulation and it may be resumed from the same point by again pressing the **H** key.

6.3 Birds have Individual Destinations or Follow a Leader:

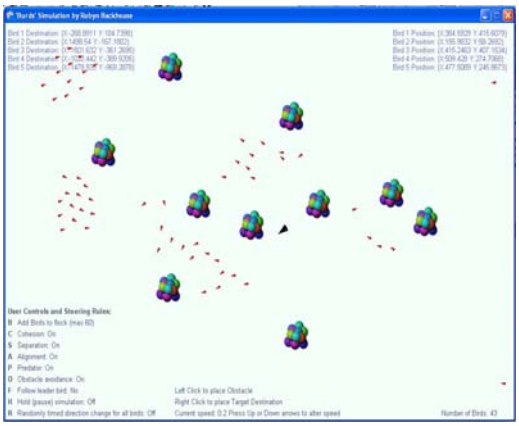
This option may be switched by pressing the **F** key on the keyboard. This will trigger a Boolean variable to change ensuring the correct bird "update" method is called each time. Default is set to birds each having individual destinations as this is far more realistic than following a leader.

6.4 Speed at which Birds Move:

Speed of birds movement can be increased or decreased using the **Up** or **Down** arrow keys on the keyboard, however the default value of 0.2 provides the most realistic and visually pleasing result for the bird's movement.

6.5 User Placed Obstacles:

The user may place up to total of 10 obstacles on the screen by left clicking the mouse button in the desired location. There is a minimal time delay set between obstacle placement due to the rapidity at which the programme runs to ensure only one obstacle is placed per click.



6.6 Random Direction Changes:

The user has an option of allowing randomly timed destination changes to all birds. This happens simultaneously to all birds (it is a single pass over the flock array, but as far as the user can tell – it appears simultaneous), and gives every bird in the flock a new randomly chosen destination. Both the timing of the change and the actual destination are randomly chosen within set limits.

As real birds appear to bank and turn as they fly, often for no reason apparent to the human observer, this option attempts to mimic such behaviour. It is turned on and off by using the **R** key on the keyboard, and is Off by default.

6.7 Predator Bird Option:

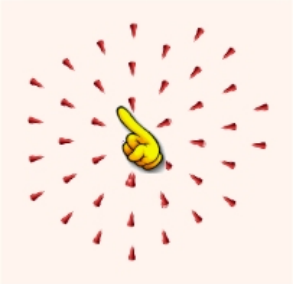
A predator bird may be enabled and disabled by pressing the **P** key. This will introduce a predator which chases the flock birds causing them to flee from it by adding the Predator Avoidance Rule to their steering behaviours. If it catches one it "eats" it and the bird is removed from the simulation. The predator is ON by default.

6.8 Addition of More Birds to Flock:

As the predator eats birds the flock grows smaller. More birds may be added by pressing the **B** key up to a maximum flock size of 60 birds (a flock of greater size suffered from poor performance). Birds will be added in groups of ten at a time.

6.9 User Selected Destination for All Birds:

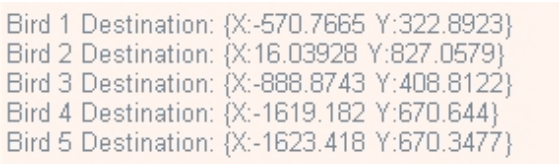
User may right-mouse-click anywhere on screen to make this the destination for all birds. Due to adherence to steering rules birds may change direction away from this point very quickly, or they may move towards that point and actually reach it, but all birds will be given that destination initially. Holding down the right mouse button will force them to hold it as their destination and they gather into a group around it.

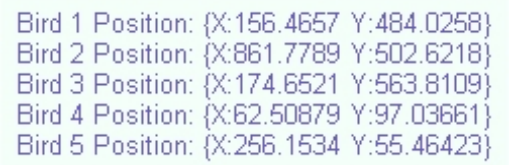


Click and hold right mouse key to force compliance

6.10 Data Visualisation:

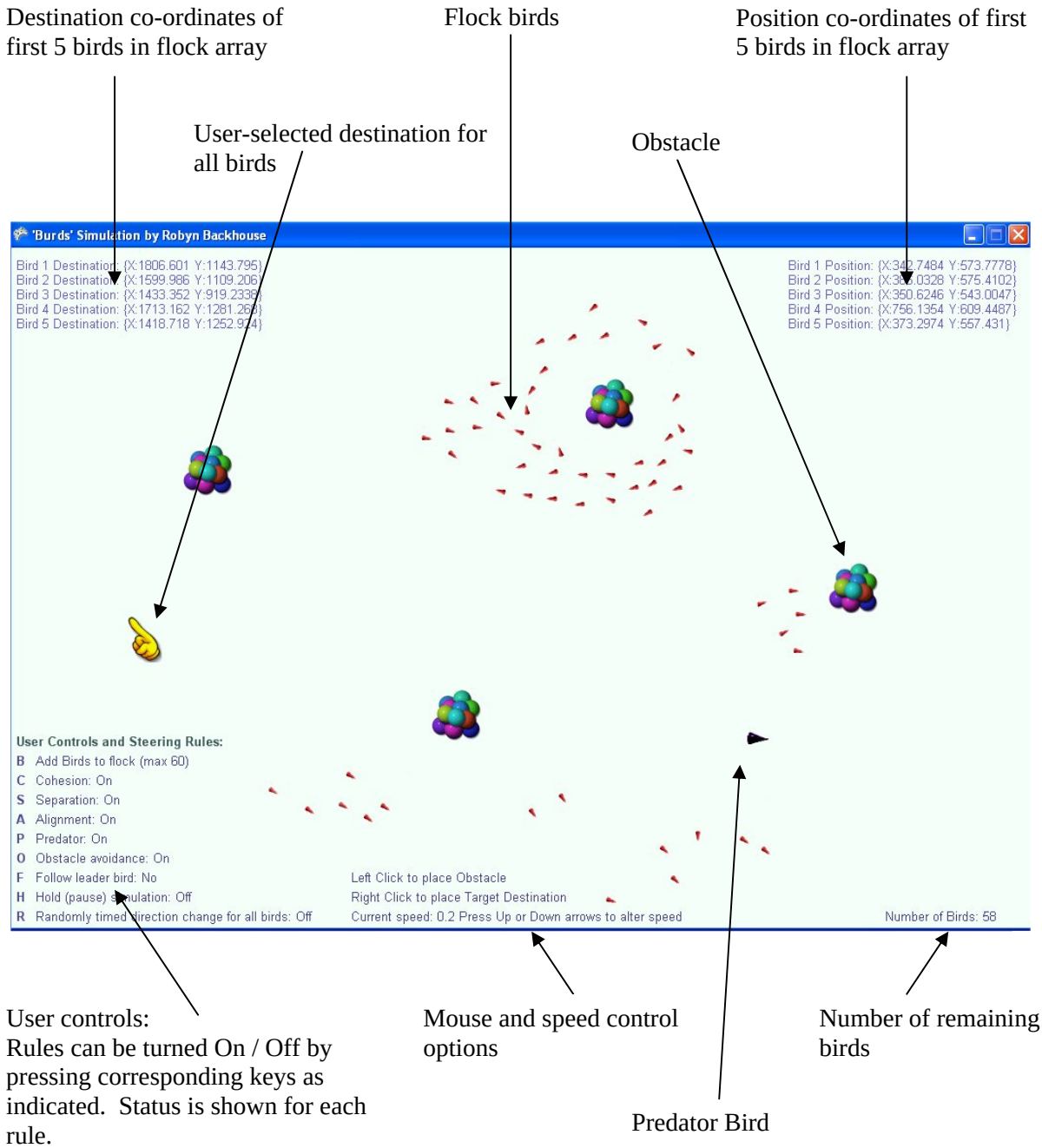
The current position and destination are shown for the first five birds in the flock to give a general picture of the vector values in real time, although they change to quickly to read accurately with the human eye. General trends may be observed.





As birds are "eaten" by the predator they are removed from the flock array, which is resized accordingly, so if one of the first five is removed it is replaced with another from the current array. Number of birds is also shown in real time, and this decreases as they are eaten by the predator bird and increases when more are added by the user.

6.11 Screen Explained:



7: RESULTS:

7.1 Emergent Behaviour with Steering Rules:

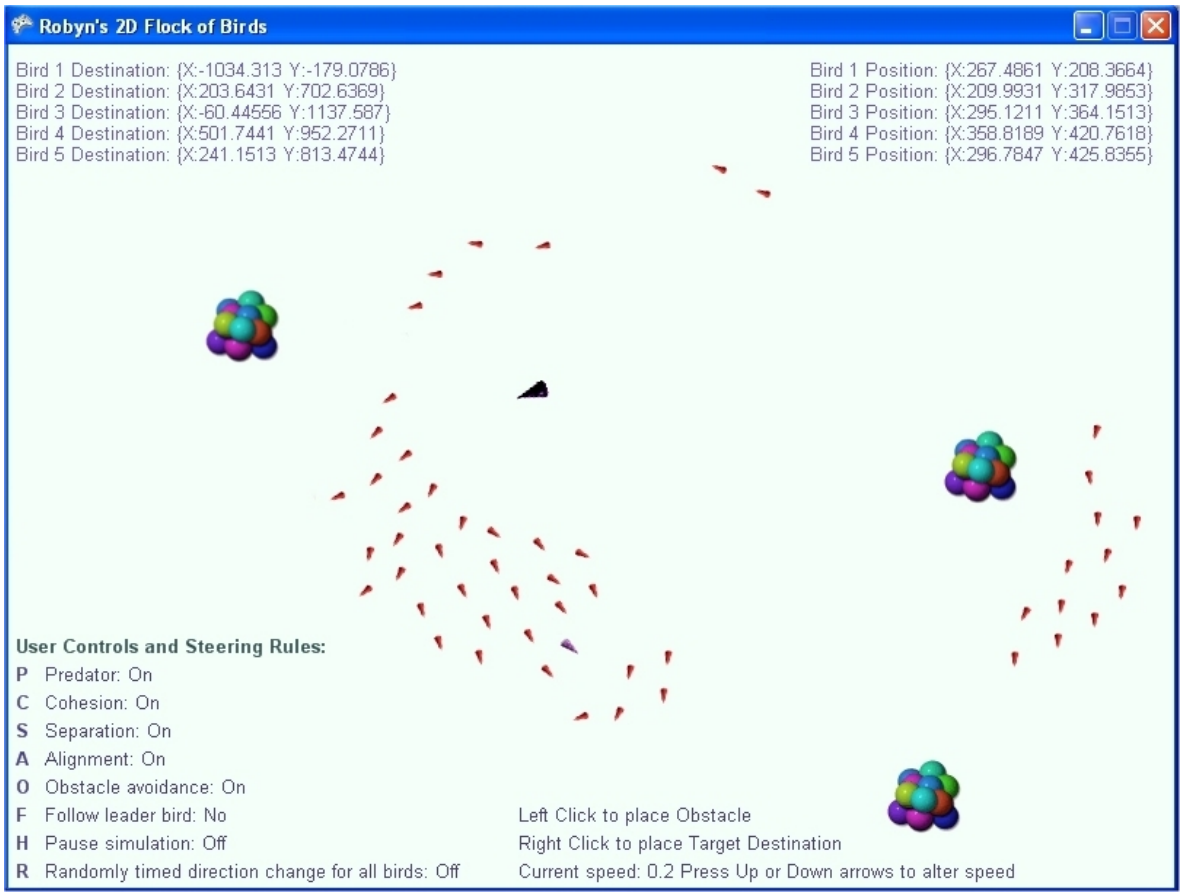
Below are a collection of screen-shots using different combinations of steering rules to illustrate their effect. For some of the following screen shots the world size was decreased for greater clarity of the actual birds, while Bird Position plus instructions for user placement of obstacles and user selected target position were disabled temporarily.

7.1.1 **All Steering Rules ON** (Cohesion, Separation, Alignment, Obstacle Avoidance):

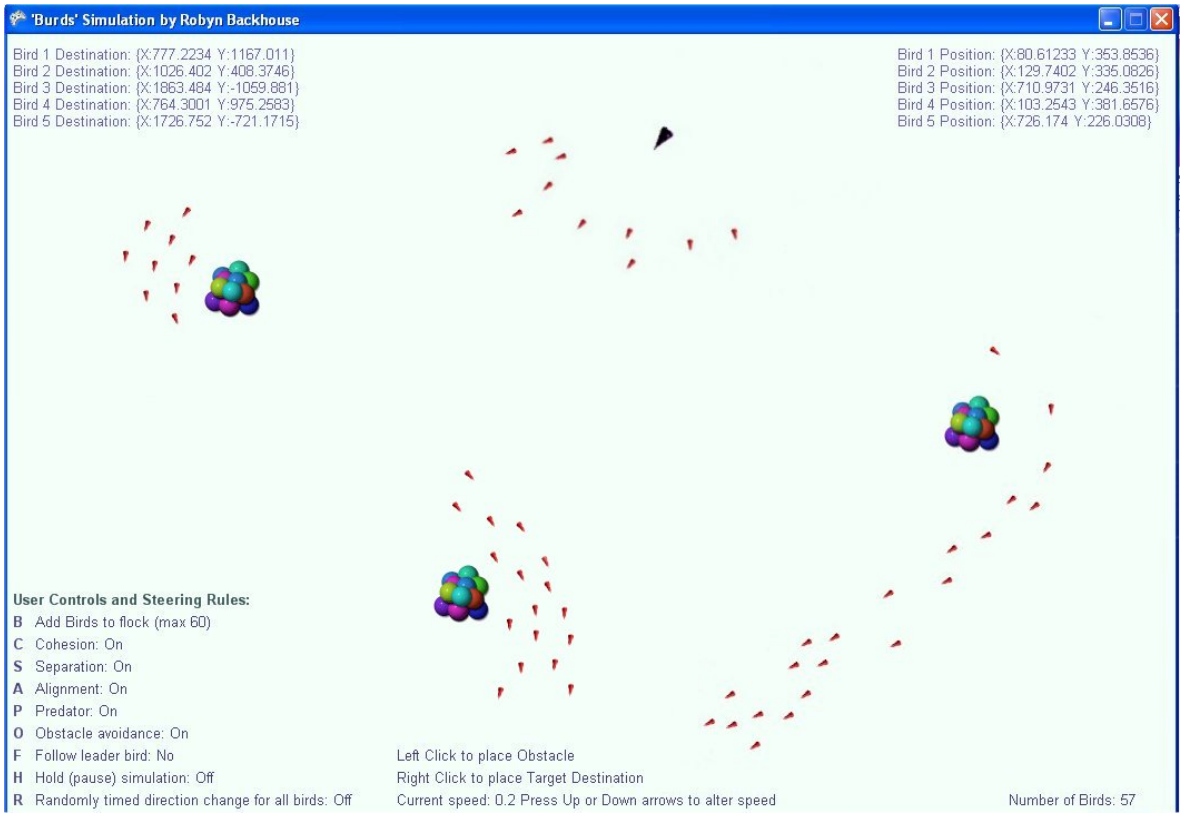
Birds flock together well when all behaviour rules are on moving with a fluid flowing motion. Cohesion pulls them together into a flock, while Separation keeps them from hitting each other, although occasional collisions do occur. Alignment ensures that they move in the average direction of their neighbours, and obstacle avoidance appears to be very effective. It is extremely rare that they collide with static obstacles, even when being "chased" by the predator. The simulation appears very effective with individual rules working well and combining to provide the desired behaviour of birds in flight.

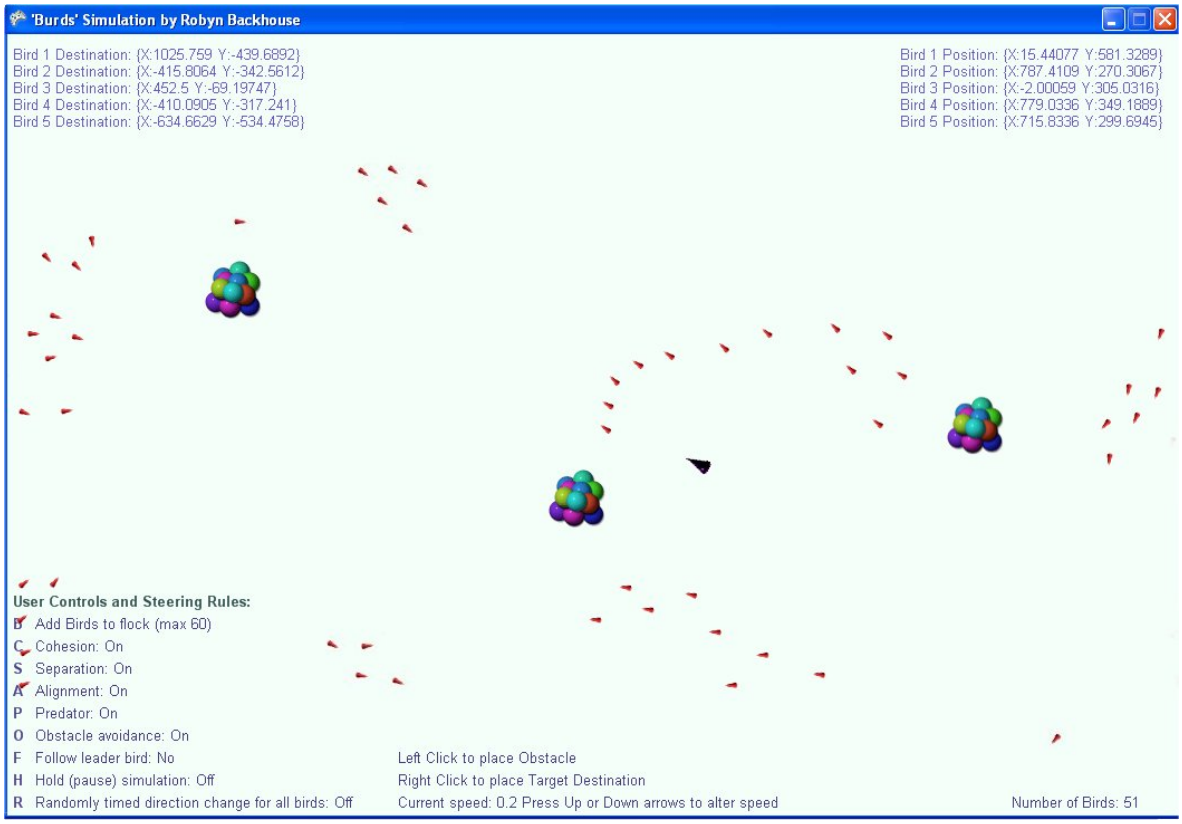
Below are some screen shots of the simulation working with all steering rules on showing a nice flowing flocking behaviour, both with and without a predator. Birds have individual destinations.

Predator Present:

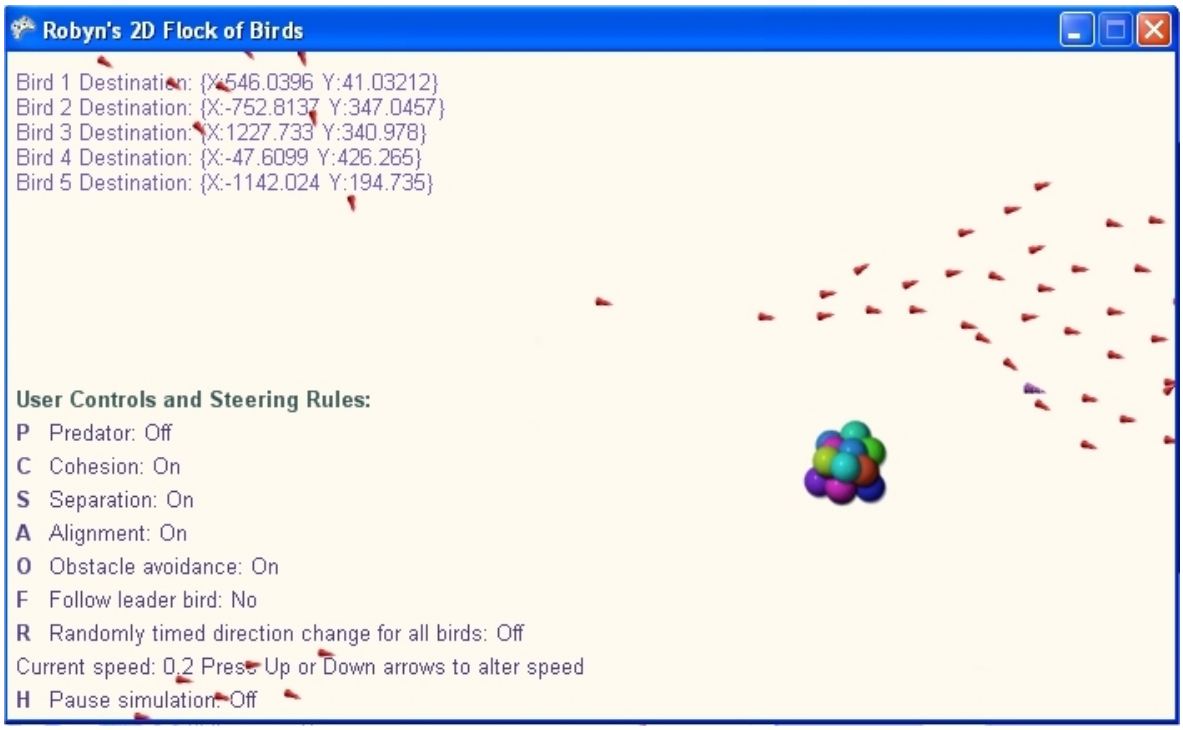


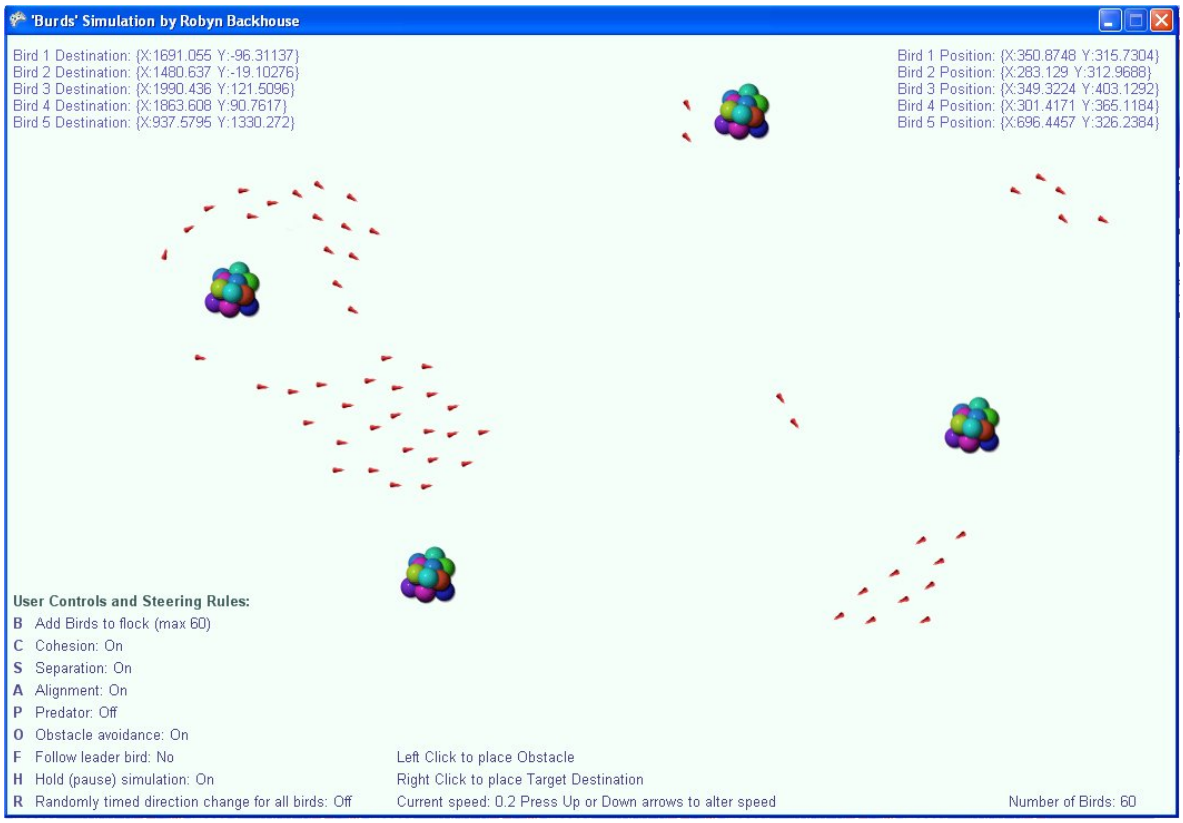
Predator Present:





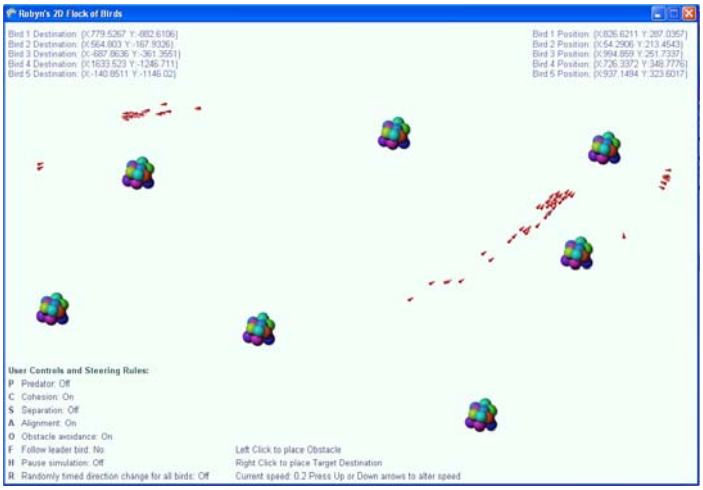
Predator Not Present:





7.1.2 Cohesion and Separation Rules:

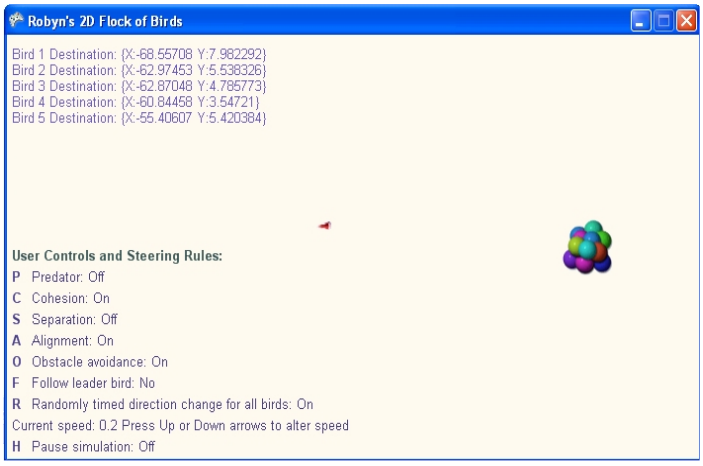
The three diagrams on this page show the effects of the Cohesion Rule, but with the Separation Rule Off. Alignment rule is On with these shots.



The Cohesion rule pulls the birds together very effectively to the extent that they appear as one bird very quickly, sharing the same position co-ordinates and moving as a single entity as far as the observer is concerned.

They not only collide but they move on top of each other, clumping together as closely as possible, and moving about in this position. The Separation Rule prevents this, keeping them apart.

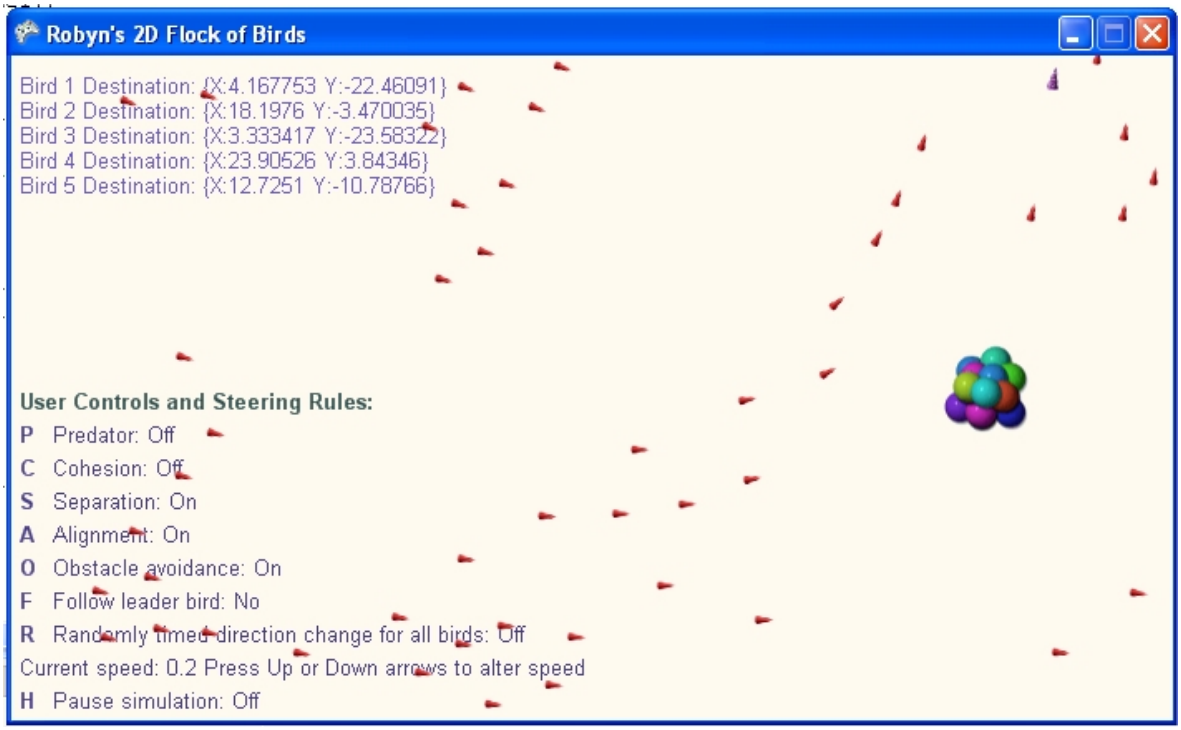
This diagram (right) actually shows 50 birds grouped so tightly that they appear as one with the Cohesion Rule on and the Separation Rule off.



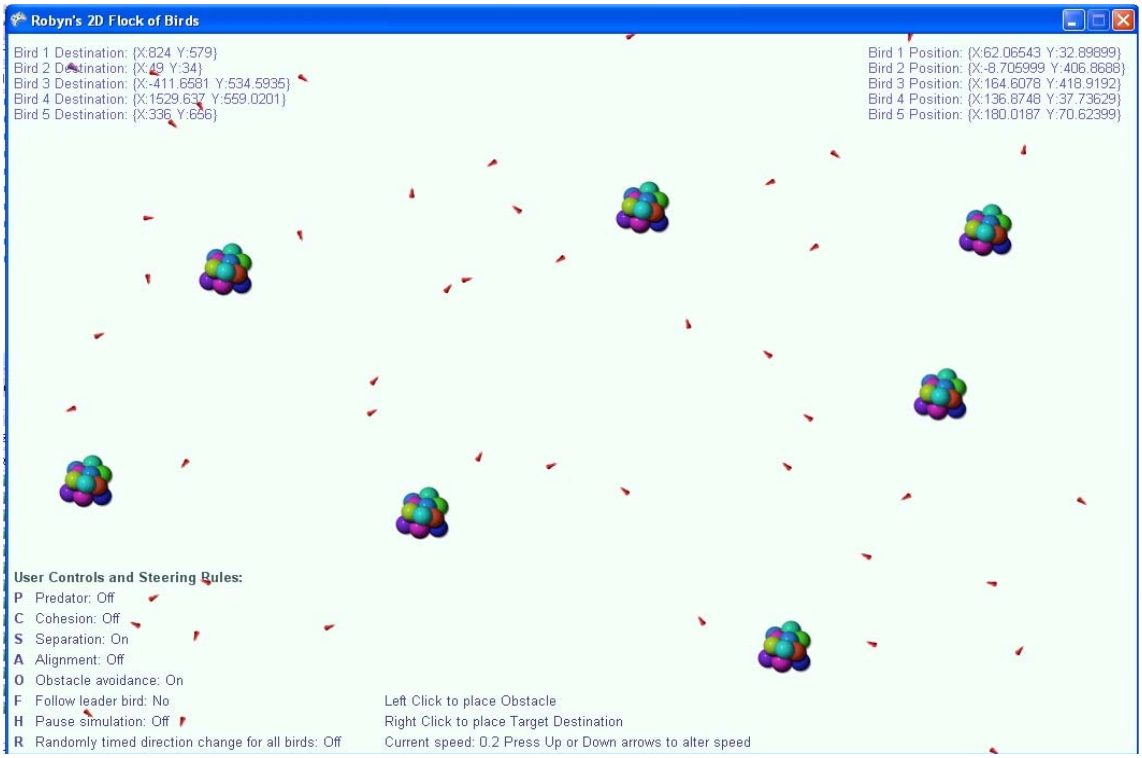
The Separation Rule aims to maintain a minimum set distance between birds at all times preventing collisions. The effect of the Separation rule is immediately apparent when it is turned off as the birds quickly clump together as shown in the preceding diagrams.

With the Cohesion Rule off the birds no longer pull together as is shown in the following diagrams, however the Separation Rule prevents collisions by maintaining a minimum distance between the birds.

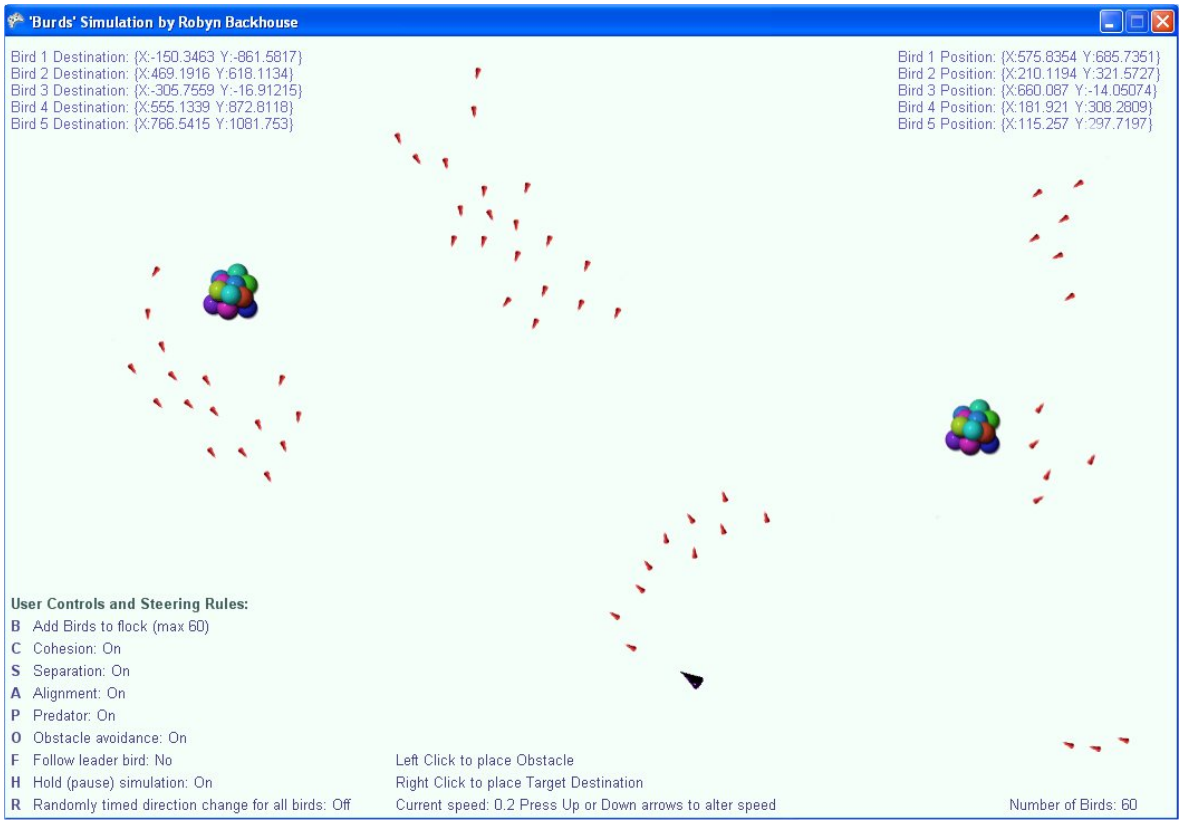
The first shows Cohesion off, but Separation and Alignment still active. The birds continue to move in the same overall direction in a flowing organised manner which is caused by the Alignment rule, however they make no attempt to remain together in a flock without the Cohesion Rule. The Separation Rule maintains a distance between the birds.



In this second diagram (following page) the Alignment rule is also deactivated, with only the Separation rule active. As can be seen the birds do not move towards each other or towards a common destination, but simply move independently around the screen in a disorganised manner. Although it cannot be seen from a static diagram, the birds each travel in a straight line without Cohesion or Alignment Rules unless threatened with collision.



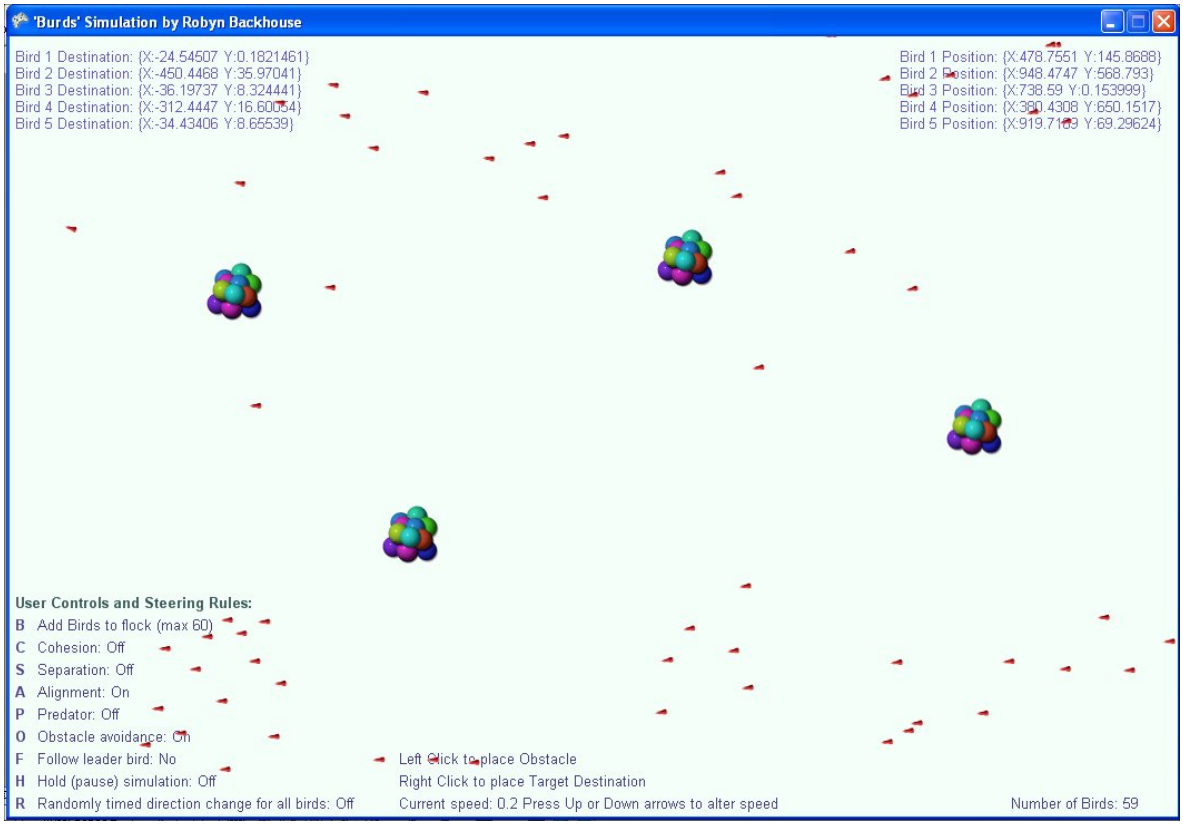
The final diagram below shows the Cohesion and Separation Rules both active, along with the Alignment Rule, Predator Avoidance and Obstacle Avoidance Rules. As may be seen the birds move together forming a flock while maintaining a minimum distance between each other avoiding collisions.



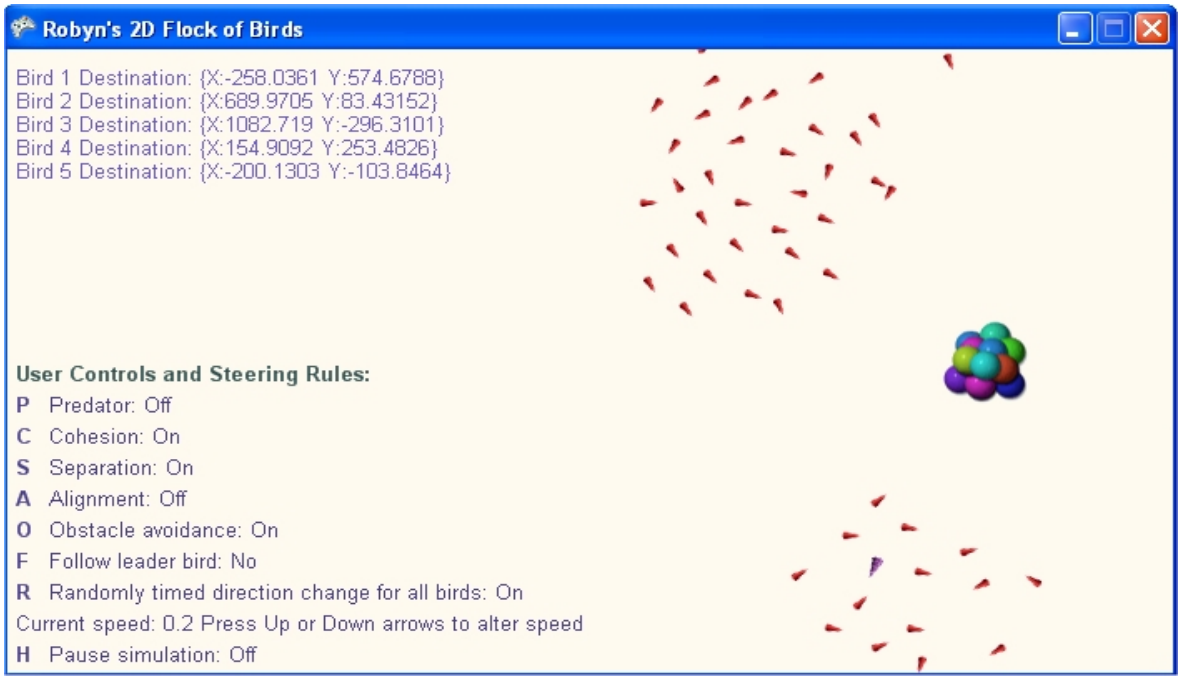
7.1.3 Alignment Rule:

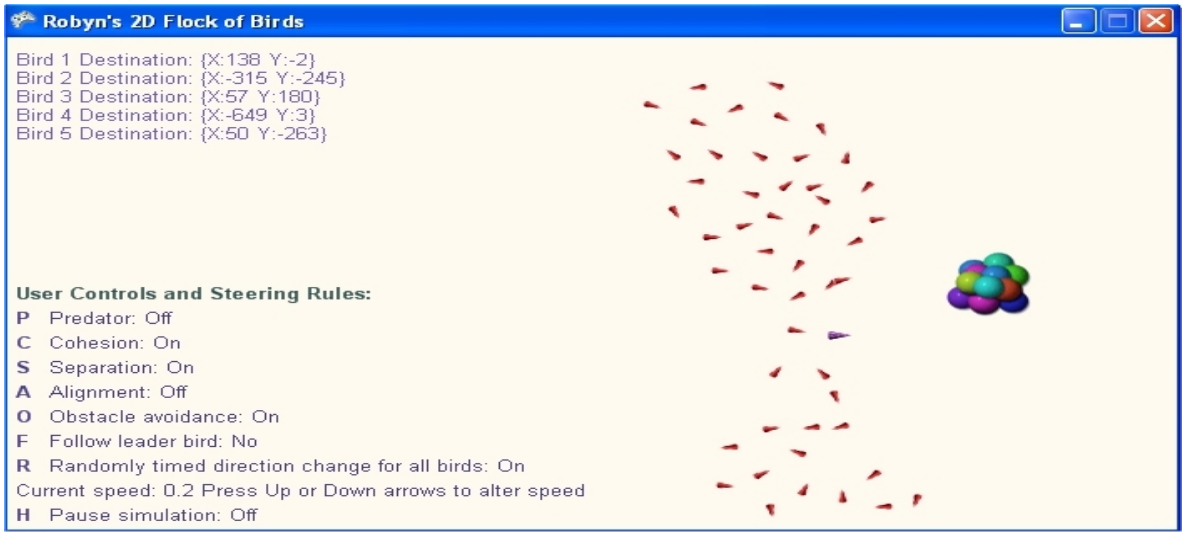
The Alignment Rule makes the birds align their destinations with the destinations of their neighbours so that they move in the same overall direction.

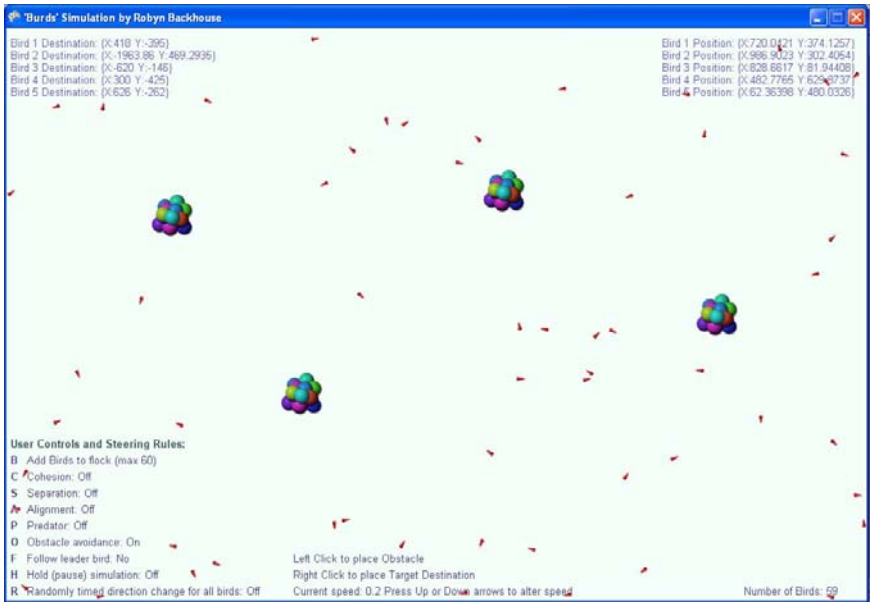
With the Alignment Rule On, but the Cohesion and Separation rules Off, the birds align their destinations to that of their neighbours and move very slowly in a straight line towards those co-ordinates. Without Cohesion or Separation they do not avoid collisions with each other, nor do they pull together to form a group. Unless an obstacle is encountered the birds do not veer from their current path once they have aligned with others but simply continue to move towards the same common direction.



When the Alignment rule is Off the birds remain in a flock staying together and avoiding collisions (with Cohesion and Separation ON), but they do not align their destinations with that of the their neighbours. This results in them behaving much like moths around a candle with no obvious purpose. The group as a whole still moves around the screen, but individuals continually "swirl" around each other within the group, not pointing towards a common heading, as shown in the following diagrams.

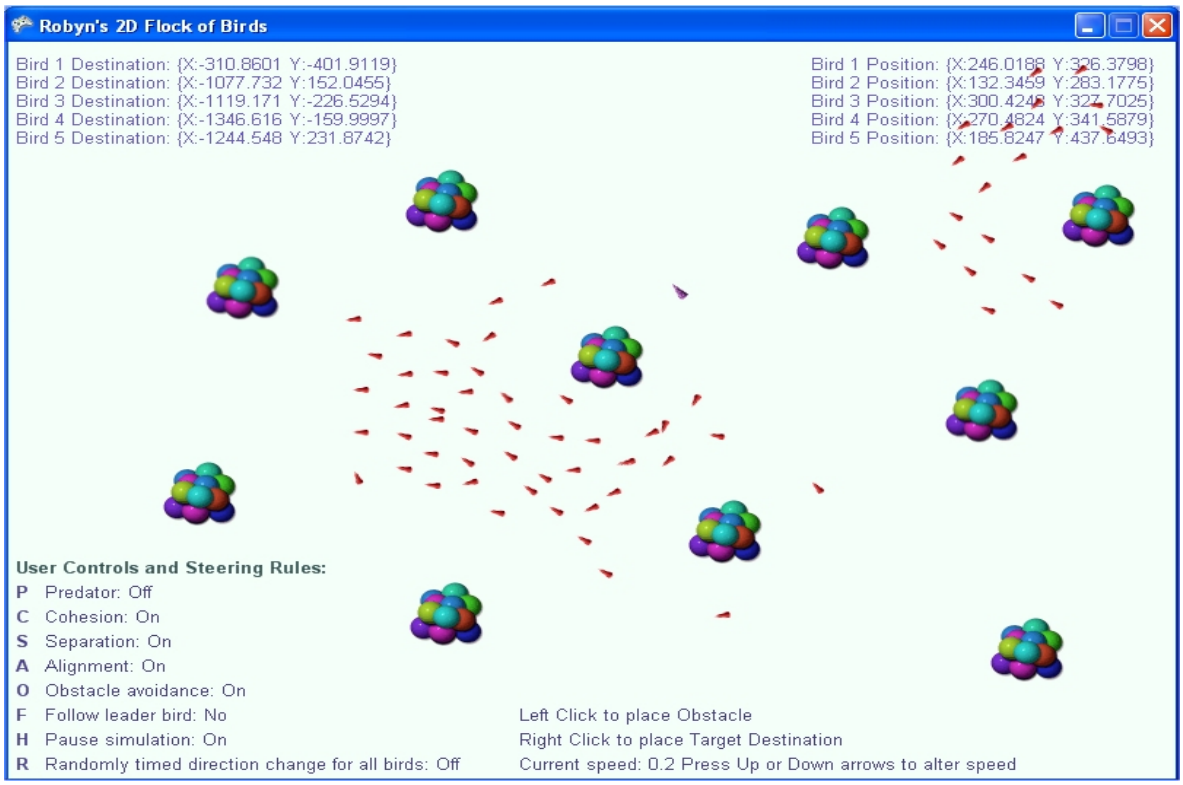






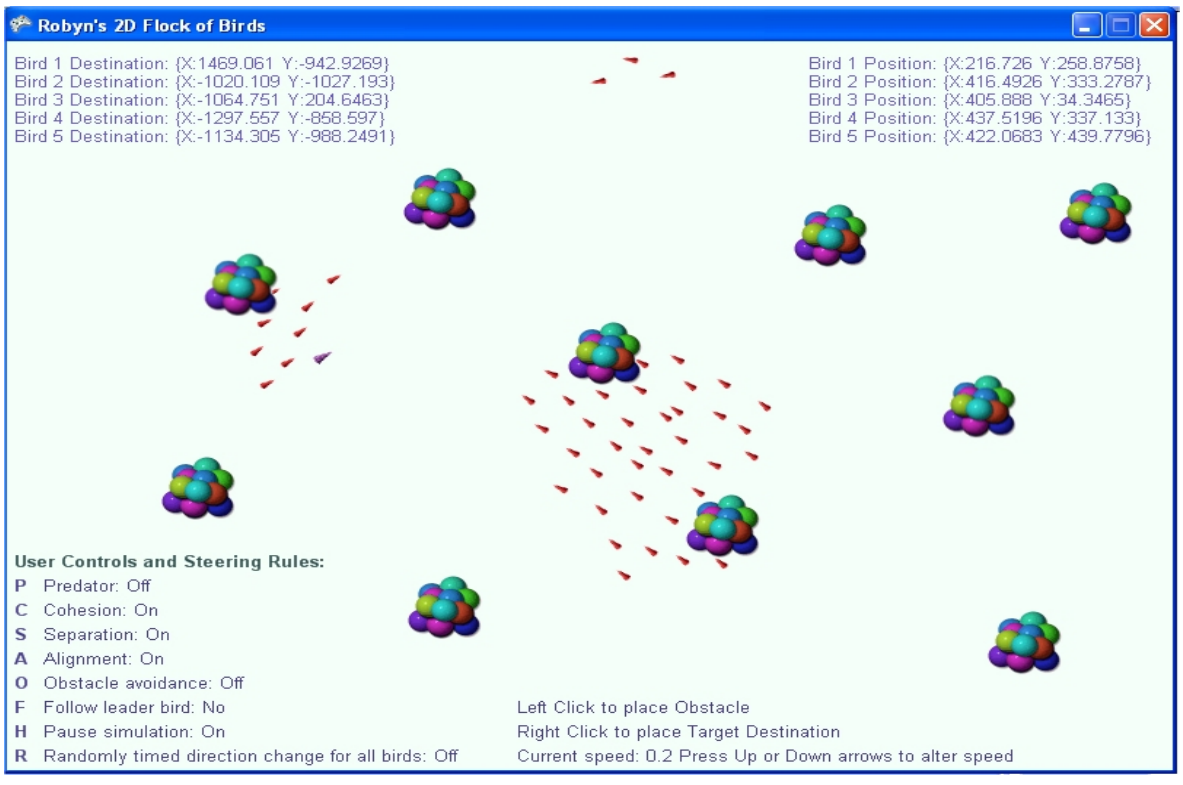
With all rules off the birds simply fly in random directions with no regard for each other at all.

7.1.4 Obstacle Avoidance Rule:



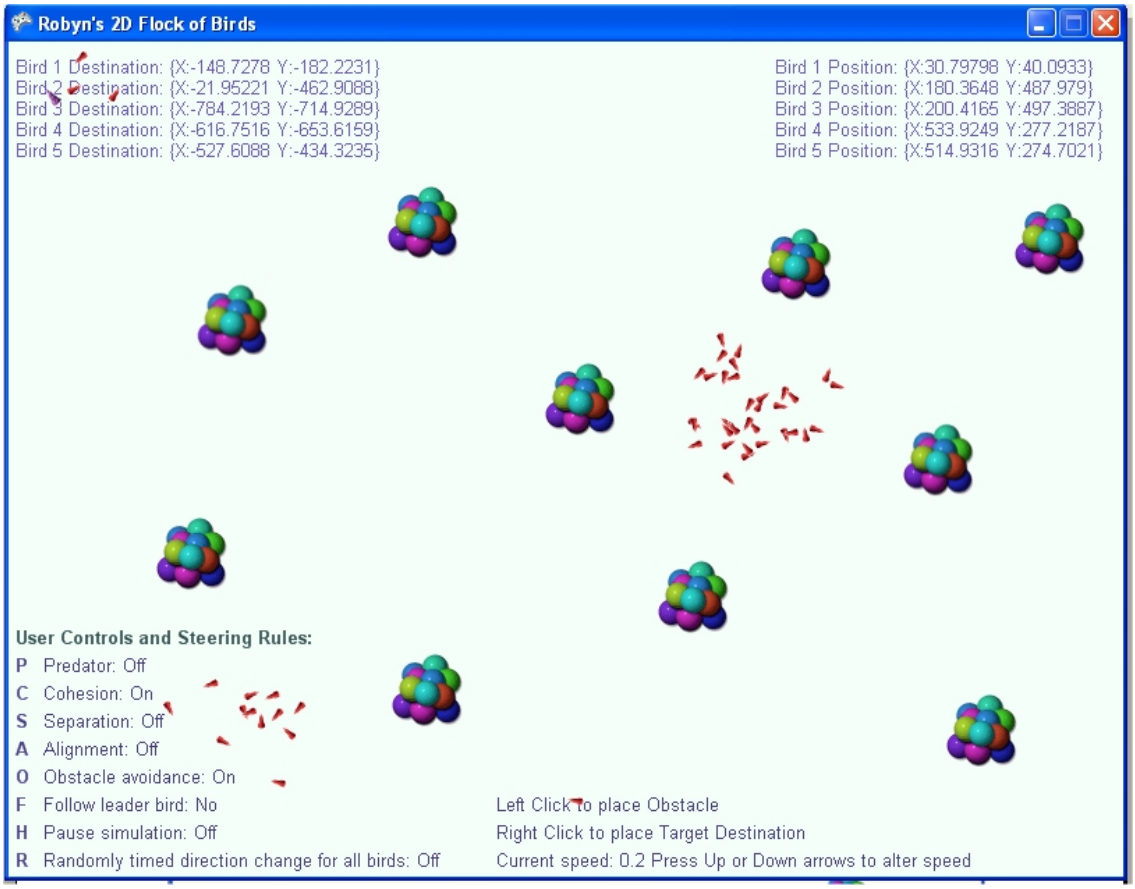
When the Obstacle Avoidance Rule is turned On birds keep a distance from the obstacles, moving around them and rarely colliding with them. Occasional collisions occur when large groups of birds are being chased by the predator and they are squeezed between them but otherwise the rule prevents collisions well. When it is Off birds simply ignore the presence of obstacles.

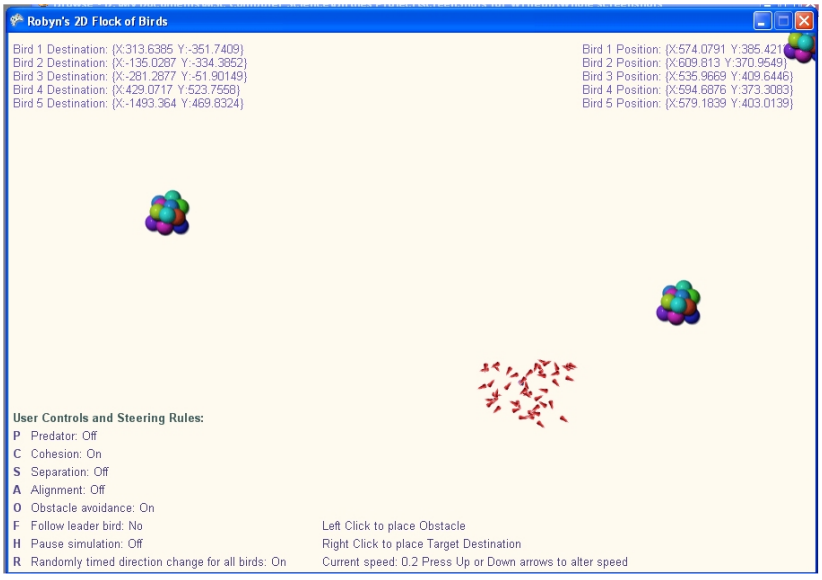
Above screenshot: Obstacle Avoidance ON. Below: Obstacle Avoidance OFF.



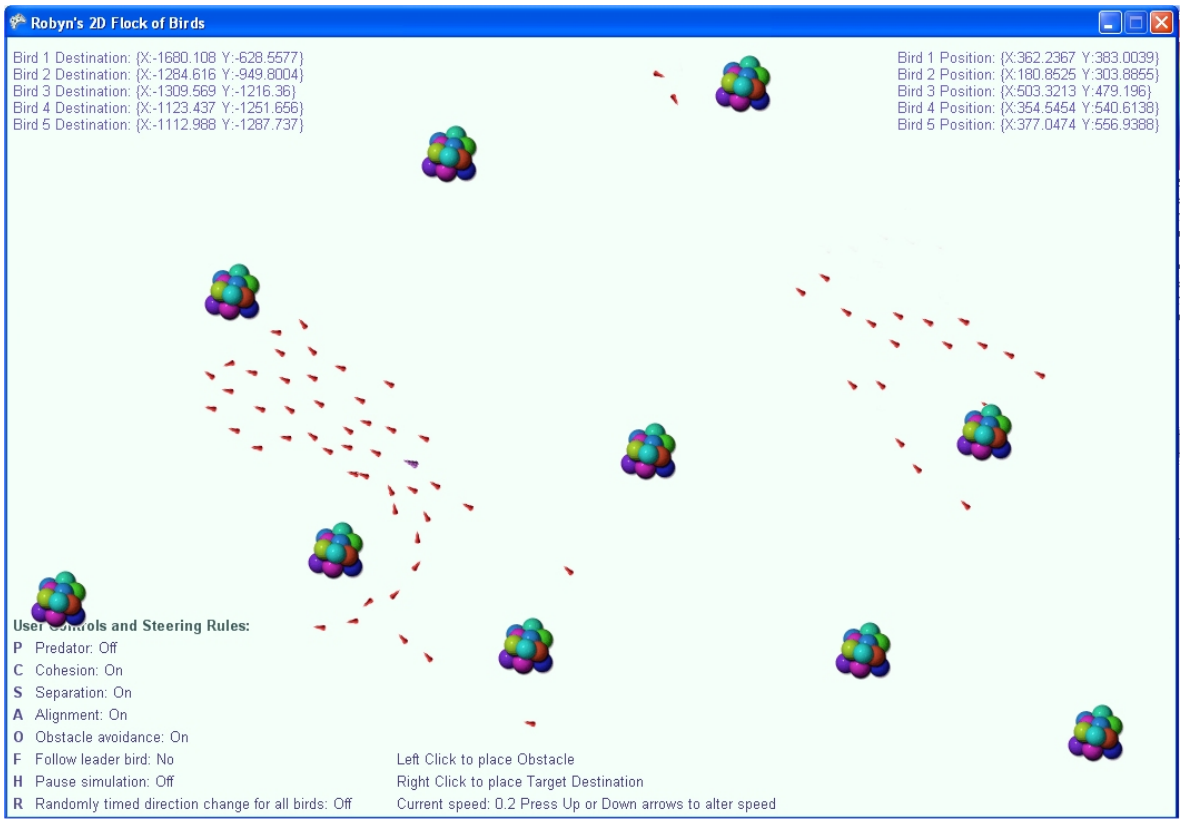
7.1.5 **Cohesion and Obstacle Avoidance Rules Only** (No Separation or Alignment)**:**

With only the Cohesion and Obstacle Avoidance rules the birds mill around in a slow moving group. When only the Separation Rule is off the birds cohere until only one bird is effectively visible (see 7.1.2), however with the Alignment Rule off as well this does not happen due to the fact they are not aligning their destinations to their neighbours and so are all trying to move in different directions while remaining in a group. This results in the birds remaining in a tight group swirling around each other but not forming a smaller group than that shown in the diagram below. Without the Separation Rule they obviously collide continuously.

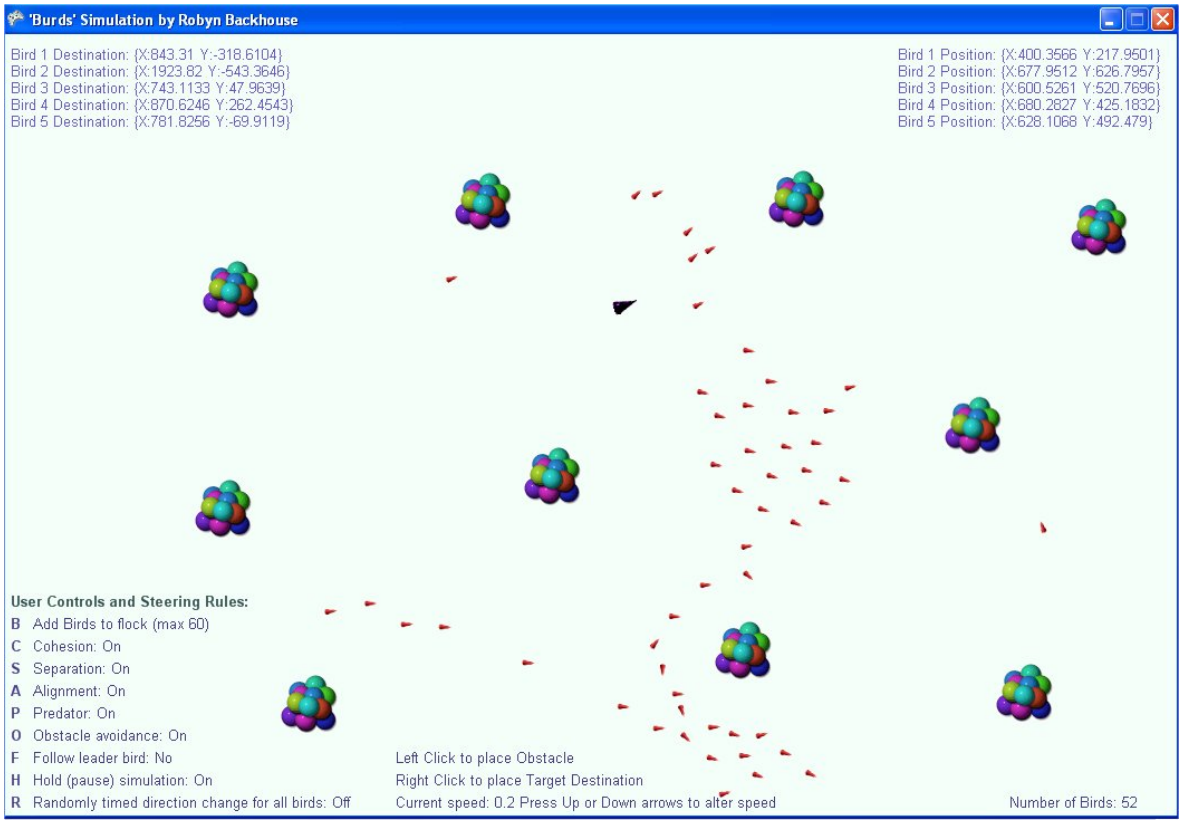




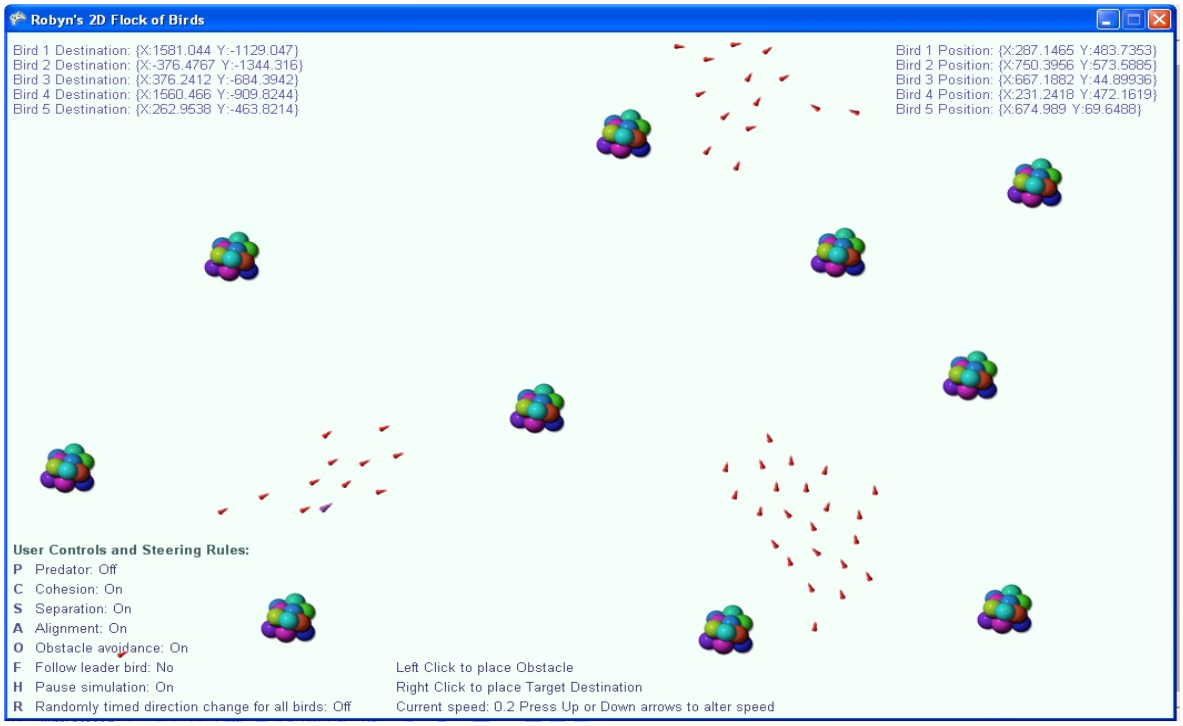
7.1.6 User Placed Obstacles:



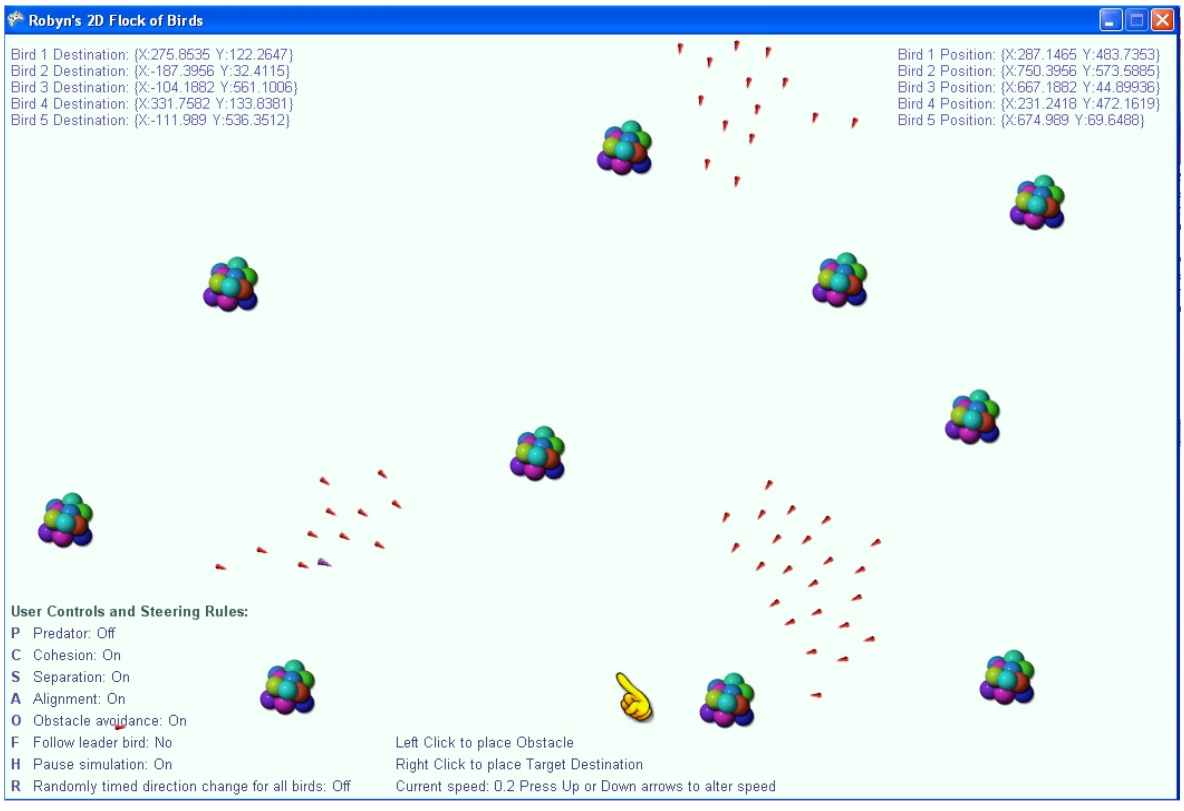
User places up to ten obstacles anywhere within the world for the birds to avoid. Birds avoid them as they do the hard coded obstacles.



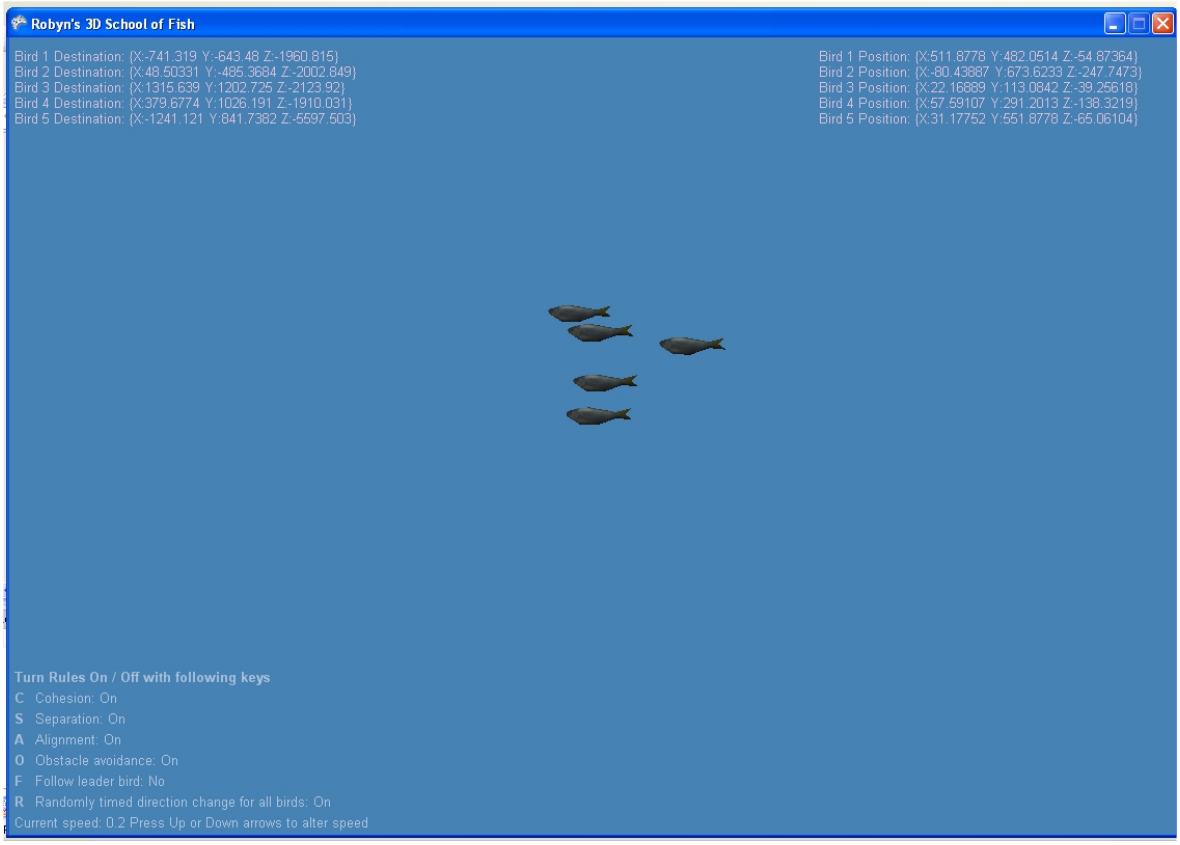
7.1.7 User Selects Target Destination for All Birds:



Simulation was paused so that birds maintained the same position for both screen shots to highlight the effect. Above the birds were following all steering rules without intervention and moving normally. Below the user targets a location on screen towards which all birds set as their destination, pointing towards this (indicated by the pointing finger). Due to compliance with steering rules birds seldom actually reach this target destination but alter course completely, however as can be clearly seen it is set initially with all birds pointing towards it.



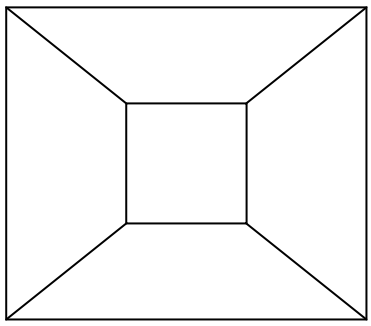
7.1.8 3D Implementation:



This paper has discussed the 2 dimensional (2D) implementation of the simulation only. The 3D implementation was very similar, using the same basic code entirely, with the addition of the extra z co-ordinate on the vectors. Initial results were promising although the mathematical formula's have not yet been fully updated as the additional co-ordinate requires significant alterations to ensure correct functioning. This remains an area for future development.

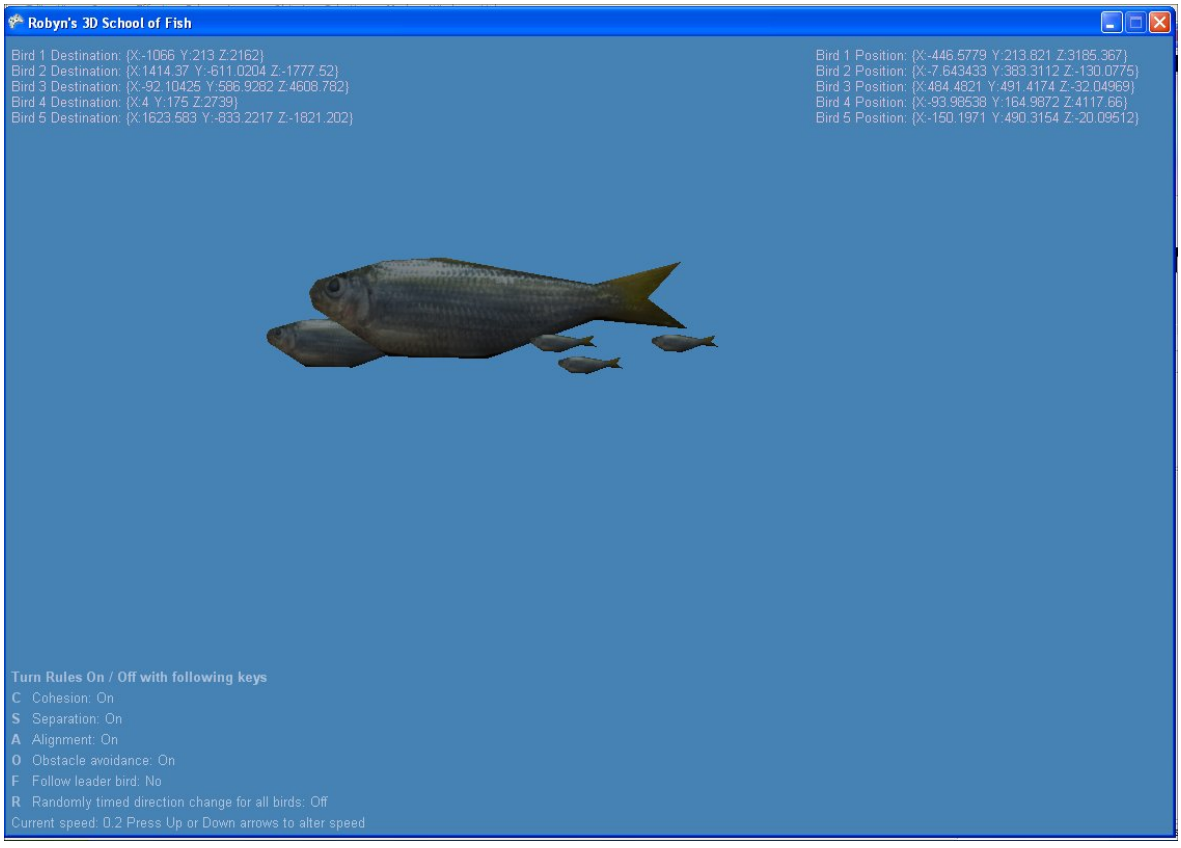
The creation of a 3D model is an entirely different area of study and as such was outside the remit of this project. Pre-made freely available models downloaded from the internet [TUR08, DUC08] were therefore used, however choice was extremely limited resulting in a school of fish instead of flocking birds (as a suitable bird model could not be found), and the obstacle used was a balloon [DUC08].

For a reason as yet unidentified the Obstacles are not being drawn in the 3D version. The Draw function is identical to that of the Bird Draw function which works well, and is being called by the same calling code as the 2D version which appears robust. The problem does not appear to lie with the model itself, as both fish and balloon models work when applied to the "Bird" objects, but neither work when applied to the Obstacle object. Complete absence of the bird array makes no difference.



The perspective of the world could be improved by rotating the camera angle or moving the viewer "into" the simulation. At present the perspective of the 3D world appears as a restricted sized box with the viewer external to the world, appearing large at the front but becoming narrower towards the back. The Richmond simulation [RIC07] appears to present a world of unrestricted size with the birds free to "fly" freely as they do in nature. This effect could be achieved by placing the viewer in the centre of the box instead of at an outer side of it, and thereby surrounded completely by the world.

The user placed obstacle option and user selected target were removed from the 3D implementation due to the difficulty in "placing" the z co-ordinate using a mouse, which is inherently restricted to 2D motion. An option would be to put a randomly chosen z co-ordinate onto the selected 2D vector, however at present this is not realistic due to the perspective of the 3D world.

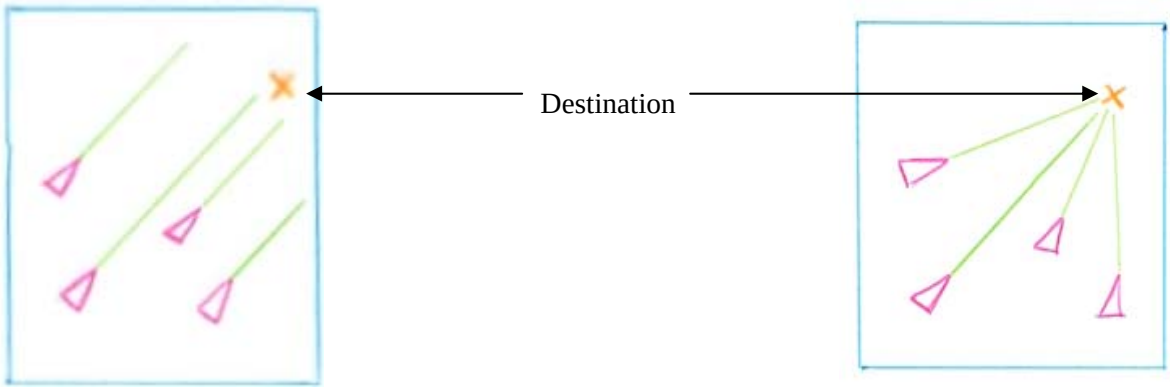


7.2: Problems Identified:

During the development of this simulation many problems were encountered, most of which were overcome. Below is a brief discussion of some of the more notable ones.

7.2.1 Shared Destination and Parallel Motion:

When birds shared a common destination they all originally moved at the same angle on the screen rather than travelling towards the same point. This was due to the fact that the destination co-ordinates are always relative to the actual bird – the bird always "thinks" of itself as having co-ordinates (0, 0) as far as any calculations are concerned. For example if the shared destination was (200, 300), then all birds would travel in the direction of (200, 300) relative to their own position and not to the actual co-ordinates of (200, 300) of their world. This also meant that birds did not appear to continually evaluate their neighbours positions within the flock and correct their own headings accordingly, but rather simply joined the group and remained in exactly the same position relative to the others. This resulted in a very static and artificial looking movement and did not prevent them from colliding with each other.



Birds share common destination, which is a co-ordinate point in relation to each bird.

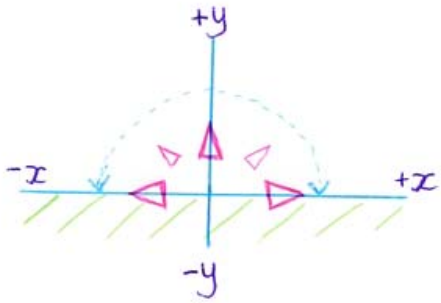
Birds have individual destinations relative to the exact destination.

It appears that LaLena's model [LAL08] suffers from the same problem with all birds in a flock moving across the screen at the same angle rather than to the same destination point. As the code is unavailable to view it is unknown whether the cause of this behaviour is the same, however when LaLena's birds are "chased" by a predator they do scatter in different directions, suggestive of individual destinations.

Sharing of a common destination is quite artificial as far as real birds are concerned. In nature there is no set path birds must follow, but rather they each have their own individual destinations which are influenced heavily by other nearby birds. This problem of parallel movement was overcome by

giving the birds their own individual headings, more like real birds, and calculating destination co-ordinates relative to the birds own position rather than simply giving it a set of co-ordinates of the world. It remains effective even if all birds have the same ultimate destination (as when the user clicks to target a position on screen), as that point will be calculated relative to each individual bird from its current position rather than being the same co-ordinates applied to each bird. This overcame the problem of their parallel movement and birds began to move in a much more realistic, flowing manner and head towards their actual destination.

7.2.2 Birds Not Facing Direction of Travel:

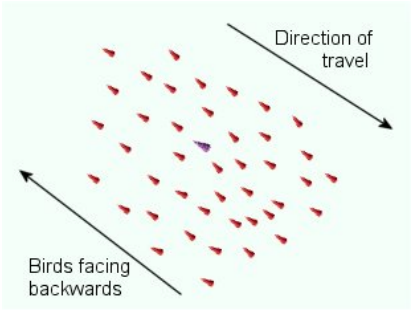


One of the most enduring problems was that of facing the birds in the direction in which they were travelling (pointing forwards). Angle of direction was originally calculated using sine values, however this produced disastrous results. Birds within the same flock all pointed in the same direction as each other, and this direction changed continuously according to their path of movement, however it was rarely in the actual direction of travel. The direction they faced appeared reasonably random, but was always pointing to a positive y co-ordinate (i.e. upwards) regardless of their direction of travel. The x co-ordinate ranged between both positive and negative (see diagram). They also frequently vanished from the middle of the window, reappearing after moments further along their path. Their positions and movements appeared to remain correct when they reappeared, however they were not being drawn on screen for periods of time.

In order to calculate the sine value, the length of the Opposite side is divided by the length of the Hypotenuse on a right angle triangle. The hypotenuse is essentially the Euclidean distance between the bird and it's destination, so this was easily calculated. The reason the birds always faced towards a positive y value is possibly due to the Euclidean distance function always returning a positive value, although this was not identified at the time and so was not explored for confirmation. The Opposite side was given the value of the x co-ordinate of the birds' destination, and so was quite possibly incorrect. Possible explanations for the birds frequently not being drawn on screen may be that division by zero was not prevented at that stage, or the incorrect figures were simply out of range (i.e. below zero or greater than 360^{o}).

Following many unsuccessful attempts at correcting this problem it was decided to try the tangent of the angle instead of the sine, as this uses only the x and y values and was therefore a more appropriate choice (see section 4.2.7.2). This produced very stable results although birds were still

facing the wrong way, however it appeared no longer random. It was immediately obvious that there was a 90° error, which was correctly simply by subtracting 90° from the angle in every case.



This resulted in the birds facing forwards when the x value was negative, but facing exactly backwards when the x value was positive. Once this was identified it was eventually resolved by adding 180° to the value whenever the x value was positive.

The final result was successful with birds now always facing correctly towards their direction of travel.

7.2.3 User Click to Target Bird Destination:

Originally when this was implemented the birds reacted by moving towards co-ordinates very different from the selected location. This appeared somewhat random, with movement sometimes being in the correct direction, but more often being incorrect. Birds always headed to positive co-ordinates from their current position. After much investigation this turned out reasonably straightforward to overcome. The new target co-ordinates were simply being added to the current position however as the target co-ordinates were always positive, as the whole viewable area of the world is positive, then this resulted in incorrect results.

For example, in the diagram if the birds' current co-ordinates were (500, 500) and the user targets an area on screen at (200, 200), the bird would travel towards a co-ordinate position of (200, 200) from it's CURRENT location and not to that location within the world, as it views itself as (0, 0). So it would move 200 units along the x axis and 200 along the y axis from it's current location actually ending up at co-ordinates (700, 700).

The solution was to change the formula by which the bird calculates it's new position from:

Bird Destination = User Target Destination *Original formula*

To the following:

Bird Destination = User Target Destination – Current Position. *New Formula*

This overcame the problem immediately and the birds now travel in the correct direction at all times.

7.2.4 Neighbourhood of Bird:

Birds appeared "confused".

The neighbourhood of a bird was improved from including all birds within a set distance from the original bird to checking their relative positions and excluding those which were behind the bird (refer to 4.1.1). It was expected that this would improve the simulation and lead to more realistic movement within the flock, as real birds cannot see those behind them either. This however was not the case initially. When the "improvement" was implemented, the motion became much jerkier and less fluid with the birds appearing "confused" and darting in all directions within the flock when they were travelling towards the negative x co-ordinates, although remaining together as a flock. The cause for this remained unknown for some time but was eventually discovered to be a flaw in the mathematical calculations of the function which measures the angle between the birds. Once this error was corrected the birds movement became fluid and graceful when the restricted neighbourhood was implemented. The resultant motion however is not noticeably different whether the birds behind are included or excluded from the neighbourhood – both options produce the same flowing movement within the flock and neither appear to give superior results over the other.

7.2.5 Flying Continually towards Negative Co-ordinates:

When the three flocking rules were originally introduced (cohesion, separation and alignment) the birds continually flew only in one direction – towards negative x and y co-ordinates (i.e. top left corner). They also appeared to completely ignore neighbourhood distance rules. This anomaly occurred only when the cohesion rule was applied which was suggestive of this being the likely cause.

The original code was based on Parker's pseudo code in which the positions of all birds were added together, then this figure (vector) was divided by the number of birds to give the average position. The position of the original bird was subtracted from this average position, and the result was then divided by 100 to move the bird approximately 1% towards this position. After much experimentation the division by 100 was removed and the vector is now multiplied by the weight for this rule, currently set to 1. This effectively overcame the problem and the birds now travel equally in all directions.

It is likely that this occurred as a result of the way the steering rules interact, although the exact cause was not identified. The vectors returned from steering rules are simply added together so

having a disproportionately small vector (just 1% of the true vector) from the cohesion rule should not theoretically cause such an anomalous result but might simply stop the birds forming a flock as effectively. Neither was it known why the overall motion was always towards negative co-ordinates. It would be assumed that if the figure was so small it would simply be the same as turning the rule off, however this was not the case. When the rule was turned off birds moved in every direction equally as would be expected.

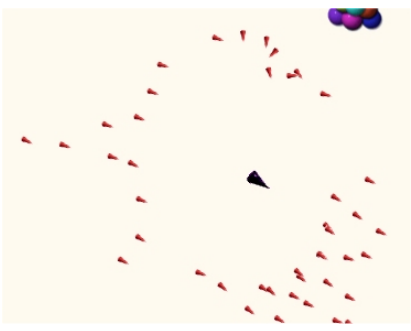
Altering the code overcame the problem with birds now moving well in all directions.

7.2.6 User Placed Obstacles:

In order to implement user placed obstacles the obstacle array had be dynamically altered which was implemented using the array list (see 7.2.8). When user placed obstacles were first introduced it was noted that birds continually appeared to become stuck or trapped at the top left corner of the obstacle, often in large groups, and seemed unable to escape. This only occurred with user placed obstacles, never with system placed obstacles. After much investigation into the cause of this it was finally discovered that it was due to the high speed at which the simulation updates, combined with the relatively slow speed of a mouse click. Each time the user placed an obstacle by clicking the mouse button, numerous obstacles were being stacked on top of each other (typically between 4 and 20), however only the top one was visible camouflaging the problem. It was observed that there were minor differences in the actual position co-ordinates of the obstacles, and that the birds must somehow have been moving between these points but then becoming unable to move away from that position due to the steering rules.

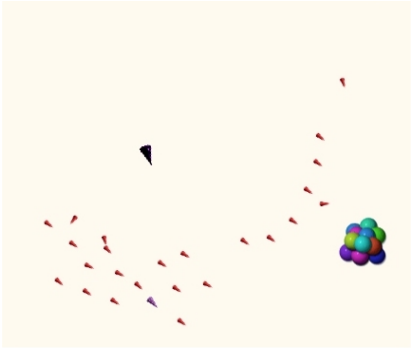
This was overcome by adding a small delay between the placement of obstacles. Once an obstacle has been placed there is a minimum time interval before another obstacle may be added. This time interval is a fraction of a second which is almost imperceptible to the user but provides an adequate interval so that multiple obstacles cannot be placed on a single mouse-click.

7.2.7 FlockBirds Surrounding a Predator:



When the predator was introduced the flock birds avoided it in the same way in which they avoid a static obstacle by maintaining a set distance away from it. They fled from it when it was chasing them but appeared to be flocking behind it and following it if there were birds beside it, as they were still flocking with these birds if they were within the neighbourhood distance (they were not flocking with the

predator, only their neighbouring FlockBirds). The result was that the predator sometimes flew surrounded by flock birds. To remedy this the position of the predator was calculated in relation to the direction in which the flock bird was facing and if the predator was ahead of the flock bird the flock bird was turned to face the opposite direction.



Despite a great deal of investigation and implementation of this solution problems persisted for some time. Large groups of birds which did not appear within range of the predator continued to follow it while spinning wildly, apparently undergoing continuous direction changes. The cause for this remained unidentified until the discovery of the aforementioned malfunctioning Bird Distance Controller function (see 5.5 and 4.2.7.4). When this was corrected the flock birds behaved in the expected manner by not following the predator and no longer spinning uncontrollably.

Birds now flee from the predator and do not follow it.

7.2.8 Dynamic Array Allocation during Runtime:

In the C# language arrays are of fixed size and cannot be dynamically altered during runtime. The only way to "change" their size is to replace the existing array with a new one, however this cannot be done directly due to restrictions within the language. In order to overcome this restriction first an array list must be created. The Array list was created initially at the beginning of the simulation and all alterations are then made directly to that list, not to the actual array. Array lists may be dynamically altered in size and hold "objects" of many types simultaneously [LIB03]. They have some functionality associated with them but this is different to the full functionality of an array so it was not possible to work directly from an Array List for running the simulation. Next, a new array of type "object" is created from the array list. Lastly this new array of "objects" is used to create a second array of "Bird" objects. The array of Birds cannot be created directly from the array list due to incompatible conversion types – there must be an intermediate array of type "object" between the Array List and the final array of Bird objects as the language does not support implicit conversion between the two. This needs to be carried out every time the array size is altered. With small arrays which do not change in size frequently performance would not be significantly affected, however larger arrays with frequent alterations may cause the programme to suffer from a dramatic reduction in performance, especially in older or slower machines. This was not seen to be a problem for this particular simulation so this method was implemented very effectively for both the Flock Bird array and the Obstacle array.

7.2.9 Difficulties Encountered with XNA:

7.2.9.1 Inability to Publish to a Standalone Application:

An unexpected difficulty encountered with the combination of Visual Studio 2008 and XNA Framework 3.0 beta is that the current versions do not fully support publication of the finished programme which means the project cannot currently be made into a standalone working executable programme, but rather can only be run within the Visual Studio/XNA environment. This functionality will reportedly be available with the final release version of XNA 3.0, but not the current beta version [XNA08]. This was not known at the time of upgrading however even the previous version does not appear to have this capability functioning correctly. This means that the only way currently to run the simulation is within the Visual Studio 2008 / XNA 3.0 Framework development environment.

7.2.9.2 User Interface:

Sliding bars to control number of birds and weights of steering rules and turning rules on and off using tick boxes would be a nice addition to the user interface and can be done with relative ease when using Visual Studio alone as it has drag and drop capability for this on a windows form. Unfortunately this is not compatible with the way in which XNA sets up the game program files, as it is a different format entirely from a windows form. Incompatibilities apparently occur if attempts are made to combine the two different approaches [MAN08], and the interface is therefore currently keyboard based which is fully supported within XNA. XNA is primarily designed for writing games for the Xbox, and therefore has limited support for more traditional computer-based control such as sliders.

8: CONCLUSION:

8.1 General Discussion:

This project presented many challenges not the least of which was learning to implement a user interface and programming moving graphics in a "game" situation, which updates frequently and responds to other objects and user intervention in real time. Many tutorials on "game programming" were completed prior to commencement of the project itself to learn about graphics programming, user interfaces and using XNA which enormously expanded my previously limited programming skills and knowledge.

When researching the works of others in the area of autonomous agents and swarm intelligence it was with some trepidation that I viewed their simulations as they appeared extremely complex, far exceeding my own programming capabilities at that time. During the progression of this simulation however this gap narrowed significantly and I now believe that while a fancier graphical user interface would enhance the aesthetic appeal of the programme enormously, the actual behaviour of the birds in my simulation exceeds most of those I previously held in awe (with the notable exception of Reynolds).

One particular area in which I regard my simulation as superior is that of collision avoidance. While it is true that my birds do still collide on occasion, moving instantly apart again, it is with markedly less frequency than other simulations which in many cases do not appear to implement any form of separation with birds continually colliding and actually "flying" continuously on top of each other making no attempt to part, or in the cases where it is implemented, colliding far more frequently than mine [GRU07, BUC05, LAL08]. Many other simulations have also not implemented any form of obstacle or predator avoidance with birds simply moving freely without interference. The inclusion of obstacles and predators creates more complex behaviour from the birds and makes the simulation more interesting for the user. It also explores behaviour which would be essential for implementation of autonomous agents into commercial applications such as military reconnaissance robots, scientific exploration robots or medical nanobots. Autonomous robots require the ability to avoid obstacles of all types, both stationary and moving, as well as each other and interacting with their environment in a predictable manner while remaining fully functional. This simulation I believe has achieved this better than most of the ones viewed although for commercial applications it would be required to have 100% reliability on obstacle avoidance and anti-collision. I believe this will be attainable with future versions and would be easier to implement with ground creatures rather than birds, as robots on the ground can simply stop moving if collision is imminent, while birds in flight cannot.

The choice of language seems to have been appropriate as does the platform of Visual Studio and XNA, however the inability to publish the simulation as a standalone application is a major drawback, as was the lack of XNA support for interface elements such as slider bars and tick-boxes for use on the PC. XNA is aimed more at development of games for the Microsoft Xbox than for PC implementations and supports far more control for the Xbox controller pad than it does for the PC keyboard and mouse. The 3D implementation has also presented difficulties as previously discussed (see 7.1.8) and remains an area for future development.

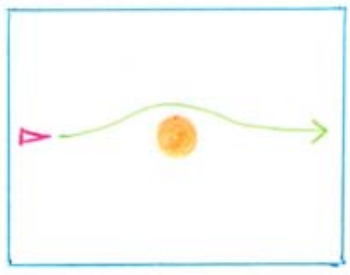
The original aims for this project were all fulfilled. Primary objectives were to provide a 2D or 3D simulation of a flock of birds moving around in a self-organising manner according to a set of attraction / repulsion rules. Additional objectives were the inclusion of obstacles and / or predators. All of these aims have been successfully implemented providing the desired behaviour and results.

Overall the project has produced successful results fulfilling all it's aims and presenting a simulation of birds flocking with relatively realistic movement. The resulting simulation has far exceeded my own expectations. Behaviour and movement of the birds is pleasing as they move with a fluidity and grace not dissimilar to that of real birds moving in a flock, which was the basis of the original objective - to create an artificial simulation of the movement and behaviour of living creatures. Steering rules may be selected and deselected by the user in order to explore the birds behaviour with each individual rule, or with different combinations, and the rules appear to work as expected.

8.2 Future Enhancements:

There are several enhancements which could improve the simulation and would be implemented into future versions.

- Obstacle avoidance would be improved by maintaining the original direction of travel instead of changing direction when obstacles are encountered, with the bird steering around them and continuing to the original destination rather than changing course. The simulation by Bourg and Seemann [BOU04] was the only one reviewed which implemented this method.



- Collision avoidance would be increased to 100% with no collisions between birds or birds and obstacles occurring. At present collisions are minimal, however for commercial applications this would need to be improved.

- An option of having more than one basic type of flock bird – so there were multiple "types" of birds, which only flock with their own kind and do not mix with each other. This could be indicated by different coloured birds (e.g. flock of red birds and flock of blue birds).

- An improved user interface would enhance usability and appearance. Features could be added such as slider bars for control over number of birds, speed of movement and weighting for steering rules. Tick boxes could also be implemented to turn steering rules and options on and off instead of using the keyboard. These improvements are purely aesthetic however, and would have no bearing on the actual behaviour of the birds.

- The option of dragging the obstacles around once they are placed would be quite nice. Currently they are placed by the user and cannot be moved. To move them to different locations at present, the simulation must be restarted and new obstacles placed.

8.3 Final Note:

While there remain a few improvements and enhancements which could be made to the simulation it has been very successful overall producing good flock behaviour with birds flying in a very pleasing and natural way, avoiding obstacles and fleeing from predators. The user interface provides users with the ability to control the bird's behaviour and to examine the effects of various combinations of the steering rules. All of the major problems faced were overcome with the final code being robust and easy to understand and maintain. The test suite ensured that functions worked correctly, while code was refactored to provide optimum functionality and maintainability.

I personally have learned a great deal while carrying out the project and feel more prepared to embark on a career within the industry having completed a more substantial piece of work. The potential applications for artificial intelligence are almost endless and it is my hope that I will be able to work within this area on some of the commercial applications for autonomous robots in the future. While this project was not a new or unique addition to the field of swarm intelligence, it is hoped that it will add to the existing body of simulations available and may itself be referenced by future students who are interested in this area of study.

9: REFERENCES:

[AFP08] AFP. (2008). Automated killer robots 'threat to humanity': expert. Feb 26th 2008. [Online] Available at http://afp.google.com/article/ALeqM5gfEAWc0aBlnuw1wuEnghZup9V7yg Accessed 13th August 2008.

[BOY04] Boyd, J. E., Hushlak, G., and Jacob, C., J. (2004). SwarmArt: Interactive Art from Swarm Intelligence. MM'04, October 10-16, 2004. New York. USA. [Online] Available at http://pages.cpsc.ucalgary.ca/~jacob/ESD/Evolutionary%20and%20Swarm%20Design/Main.html Accessed April 2008.

[BOU04] Bourg, D., Seemann, G. (2004). AI for Game Developers. O'Reilly. *Simulation to accompany this book downloaded from Professor Barnes, California State University Northridge. flock2.exe [Online] Available at http://www.csun.edu/~renzo/cs565/ Accessed 29th May 2008.

[BRO00] Browning, V. (2000). Nanotechnology: Nano-medicine. [Online] Available at http://www.wildirisdesign.com/nano/nanomedicine.html Accessed 12th August 2008.

[BUC05] Buckland, M. (2005). Programming Game AI by Example. Wordware. *Simulation to accompany this book downloaded from Professor Barnes, California State University Northridge. Flocking.exe [Online] Available at http://www.csun.edu/~renzo/cs565/ Accessed 29th May 2008.

[DAV05] Davison, A. (2005). Killer Game Programming in Java. O'Reilly Media, Inc.

[DUC08] Duchamp Models. (2008). 3D Models, Textures, Animations. (Free Balloon 3D Model). [Online] Available at http://www.seemonkey.com/duchamp_models Accessed June 2008.

[EVE08] Evans, D. (2008). Robosoldier. Focus : Ethics. ITNOW: the magazine for the IT professional. The British Computer Society. May 2008. pp 6-7.

[FRA07] Fraunhofer-Gesellschaft. (2007). Electronic Nurses. [Online] Available at http://www.medicalnewstoday.com/articles/62332.php Accessed 12th August 2008.

[GAR08] Garner, D. (2008). Evolving 'Swarm' Robots Investigated By UK's University Of York. 13th March 2008. [Online] Available at http://www.medicalnewstoday.com/articles/100479.php Accessed 12th August 2008.

[GIL08] Gilliand, B. (2008). Creepy, squishy and crawly… meet the future of military robots. Science & Discovery. Metro (newspaper). Friday July 25, 2008. p 17.

[GOR05] Gorman, J. (2005). Agile.NET Development – Test-driven Development using NUnit. parlez|uml [Online] Available at http://www.parlezuml.com/tutorials/agiledotnet/tdd_nunit.pdf Accessed March 2008.

[GRU07] Grubb, T., G. (2007). Flocking Demo. Object Pascal source code for implementing flocking and formation flocking using CodeGear's Delphi. RiversoftAVG. [Online] Available at www.RiverSoftAVG.com Accessed 4th June 2008.

[JAC07] Jacob, C., Hushlak, G. (2007). Home of Swarm Art. A Collaborative Partnership between Science and Art. University of Calgary. [Online] www.swarmart.com Accessed 11th August 2008.

[JAC06] Jacob, C. (2006). Evolutionary and Swarm Design. University of Calgary. [Online] Available at http://pages.cpsc.ucalgary.ca/~jacob/ESD/Evolutionary%20and%20Swarm%20Design/Main.html Accessed 12th August 2008.

[LAL08] LaLena, M. (2008). Bird Flocking Behaviour Simulator. [Online] Available at http://www.lalena.com/AI/Flock/Flock.aspx Accessed 14th June 2008.

[LAR01] Larman, C. (2001). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR.

[LIB03] Liberty, J. (2003). Programming C#. 3rd Edition. Building .NET Applications. O'Reilly.

[MAN08] Mannock, K. (2008). Conversation between Dr Mannock and myself regarding programming in 3D and choice of platforms. Discussed benefits of XNA over lesser known platforms. 3rd June 2008. Location: DCS Birkbeck College, London.

[MAN08a] Mannock, K. (2008). Lecture on C#, Microsoft Visual Studio and .NET environment. Department of Computer Science. Birkbeck College. University of London. January 2008.

[MAN07] Mannock, K. (2007). Autonomous Agents. Ideas for MSc Projects (MSC CS). [Online] Available at http://www.dcs.bbk.ac.uk/~sven/projects/mscprojideascs.html Accessed Nov 2007.

[MIL08] Miles, R. (2008). Learn Programming Now! Microsoft XNA Game Studio 2.0. Design and create games for your Xbox 360 and your PC. Microsoft Press.

[MIL07] Miller, P. (2007). Swarm theory. National Geographic.Com. [Online] Available at http://ngm.nationalgeographic.com/ngm/0707/feature5/index.html Accessed Feb 2008.

[PAL08] Palmer, J. (2008). Smart future for swarming robots. BBC News. [Online] Available at http://news.bbc.co.uk/1/hi/technology/7549059.stm Accessed 8th August 2008.

[PAR07] Parker, C. (2007). Boids Pseudocode. [Online] Available at http://www.vergenet.net/~conrad/boids/pseudocode.html Accessed 25th April 2008.

[PRE01] Pressman, R. S. (2001). Software Engineering A Practitioners Approach. Fifth Edition. McGraw Hill.

[REE83] Reeves, W., T. (1983). Particle Systems – A Technique for Modelling a Class of Fuzzy Objects. acm Transactions on Graphics, 2(2). April 1983. Reprinted Computer Graphics, 17(3), July 1983, (acm SIGGRAPH'83 Proceedings). pp 359-376.

[REY01] Reynolds, C. W. (2001). Boids background and update. [Online] Available at http://www.red3d.com/cwr/boids/ Accessed 25th April 2008.

[REY99] Reynolds, C. W. (1999). Steering Behaviors For Autonomous Characters. In the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California. Pages 763-782.

[REY88] Reynolds, C. W. (1988). Not Bumping Into Things. Notes on "obstacle avoidance" for the course on Physically Based Modelling at SIGGRAPH 88, August 1 through 5 in Atlanta, Georgia. [Online] Available at http://www.red3d.com/cwr/nobump/nobump.html Accessed 25th April 2008.

[REY87] Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioural Model. Computer Graphics, 21 (4), SIGGRAPH'87, pp 25-34.

[RIC07] Richmond, P. (2007). Java Boids Simulator - Demonstrating Bird Flocking (updated for use with jogl JSR-231 1.1.0 Release Candidate 2 - 23Jan 07) [Online] Available at http://www.dcs.shef.ac.uk/~paul/publications/boids/index.html Accessed 14th August 2008.

[TUR08] Turbo Squid. (2008). Commercial site for downloading 3D models. [Online] Available at http://www.turbosquid.com/XNA Accessed June 2008.

[USA08] U.S. Army Research Laboratory. (2008). Micro Autonomous Systems and Technology. U.S. Army Research, Development and Engineering Command. [Online] Available at http://www.arl.army.mil/www/default.cfm?Action=93&Page=332 Accessed 13th August 2008.

[WEI08] Weisstein, E., W. (2008). "Distance" From MathWorld – A Wolfram Web Resource. [Online] Available at http://mathworld.wolfram.com/Distance.html Accessed 24th July 2008.

[WIK08] Wikipedia. (2008). Euclidean Distance. [Online] Available at http://en.wikipedia.org/wiki/Euclidean_distance Accessed 24th July 2008.

[WIK08a] Wikipedia. (2008). Swarm Intelligence. [Online] Available at http://en.wikipedia.org/wiki/Swarm_intelligence Accessed 11th August 2008.

[WIK08b] Wikipedia. (2008). Nanorobotics. [Online] Available at http://en.wikipedia.org/wiki/Nanorobotics Accessed 12th August 2008.

[WIK08c] Wikibooks. (2008). Geometry/Right Triangles and Pythagorean Theorum. [Online] Available at http://en.wikibooks.org/wiki/Geometry/Right_Triangles_and_Pythagorean_Theorem Accessed 24th August 2008.

[WIL99] Wiley, K. (1999). Flock with Obstacles. [Online] Available at http://www.cs.unm.edu/~kwiley/artificialLife/flockWithObstacles.html Accessed 20th June 2008.

[WIL99a] Wiley, K. (1999). Gnat Cloud. [Online] Available at http://www.cs.unm.edu/~kwiley/artificialLife/gnatCloud.html Accessed 20th June 2008.

[WIL99b] Wiley, K. (1999). Mega Flies. [Online] Available at http://www.cs.unm.edu/~kwiley/artificialLife.html Accessed 20th June 2008.

[XNA08] XNA Creators Club Online. (2008). [Online] Available at http://creators.xna.com/ Accessed from June 2008 onwards.

10: APPENDICES

LIST OF APPENDICES:

APPENDIX A

Quick Guide to User Control Keys:

- Cohesion: press key **C**
- Separation: press key **S**
- Alignment: press key **A**
- Obstacle avoidance: press key **O**
- Predator bird present: press key **P**
- Pause simulation (hold): press key **H**
- Random Destination Change for all birds: press key **R**
- Follow Leader or Individual Destinations: press key **F**
- Add Birds to Flock (groups of 10): press key **B**

- Increase Speed of bird's movement: press **Up arrow** key
- Decrease Speed of bird's movement: press **Down arrow** key
- Place Obstacle on screen: **left click** mouse on desired location
- Select Destination for all birds: **right click** mouse on desired location (may click and hold to force compliance)

User option keys have not been found to be case sensitive on any machines upon which the simulation has thus far been tested.

APPENDIX B

Project Proposal Form Submitted:

School of Computer Science and Information Systems

MSc Computer Science Project Form (2008)

- ## Proposal

  The student should complete parts 1 (a) and 1 (b) below. They should put their supervisor's name, in part 2 (a) below. They should send the completed electronic copy of this form to the projects tutor (mick@dcs.bbk.ac.uk) and to the course administrator (compadmin@dcs.bbk.ac.uk) no later than **Friday,16 May 2008**.

(a) Student details

Name: ***Robyn Backhouse*** *(Personal Details Removed)*	**Address:** *(Removed)*

(b) Project details

Title: Flocks of Birds
Objectives: To provide a 2D or 3D simulation of a flock of birds. Primary objective: - Flock of birds moving around in a self-organising way according to a set of attraction/repulsion rules. Additional objectives (if time permits): - Flock of birds avoiding obstacles (separation and regrouping) and/or predators.
Description: This work involves different elements: - Algorithm/Model for simulating the flock of birds' behaviour. - A graphical user interface for selecting initial inputs (parameters such as number of birds, position of obstacles, etc); - A 2D or 3D visualisation of the simulation.

Title: Flocks of Birds

Method:

The method involves:

- Study of existing flock of birds behaviour models possibly including obstacle and predator avoidance.
- Determination/Definition of the set of attraction/repulsion rules to apply to each bird (model)
- Choice of an implementation platform. Likely choice: 3DState with Microsoft Visual Studio.
- Implementation of the identified model.
- Development of the GUI for selecting initialization parameters
- 2D or 3D visualization of the simulation.

Work plan:

Preliminary lectures and choice of platform: 30th May
Study and identification of models: 15th June

First version of implementation (model): 15th July
Second Version (including GUI): 31st July
Third Version (including 2D/3D visualisation): 15th August

Test / Evaluation / Analyses of models: 31st August

Writing Report (v1): 30th June
Writing Report (v2): 31st July
Writing Report (v3): 31st August

Project Complete 24th September

College equipment required:

none

APPENDIX C

Specification for Project on Birkbeck Website:

This is the specification for the project which was on the "Ideas for MSc Projects" page of the School of Computer Science Birkbeck Website where a number of suggestions are presented for students.

"An autonomous agent is a simple entity that interacts with its environment and other autonomous agents, typically based on a simple set of rules. For example, the autonomous agents may be birds that randomly fly around a grid, but obey simple rules like avoiding bumping into each other and not flying directly behind another bird (otherwise it cannot see). It is interesting to then observe the "emergent behaviour" such as the patterns of movement. Researchers have developed very simple rules that seem to mimic the patterns seen in nature of how birds fly in formations. In this project, you will explore the various types of autonomous agents and their behaviours."

Keith Mannock [MAN07].

APPENDIX D

Code for 2D Flocking Birds:

D.1 Class: Program

```
using System;

namespace Burds2D
{
    /// Creates a new Game1 (or simulation) and once created,
    /// runs it until it is terminated by the user
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            // Creates the new simulation
            using (Game1 FlockingBirds = new Game1())
            {
                // Runs the newly created simulation
                FlockingBirds.Run();
            }
        }
    }
}
```

D.2 Class: Game1

```
using System;
using System.Collections;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    /// <summary>
    /// This is the main type for the project
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch birdBatch;
        SpriteBatch obstacleBatch;
        SpriteBatch fontBatch;
        UserTarget birdTarget;
        KeyboardState currentKeyboardState;
        SpriteFont font;
        SpriteFont fontB;

        private FlockBird[] flockArray;
        private ArrayList flockBirdArrayList = new ArrayList();
        private object[] flockBirdArrayObject;
        private object[] obstacleArrayObject;
        private Obstacle[] obstacleArray;
        private ArrayList obstacleArrayList = new ArrayList();
        private PredatorBird predator;

        private int numberOfBirds = 50;
        private int numberOfObstacles = 2;
        //Controls the speed of the birds
        private float speedRestricter = 0.2f;
        private string flockBirdAssetName = "flockBird";
        private string leaderAssetName = "leaderBird";
        private string obstacleAssetName = "bubbleObstacle";
        private string predatorAssetName = "predatorBird";
        //Time taken from clock - used for random timing calculations
        //(clock cannot be used directly for the type of calculations)
        private int timePlayed = 0;
        private float displayWidth;     //Width of window
        private float displayHeight;    //Height of window

        //GUI controlled elements - changed during runtime by user
        private bool randomlyTimedDestChange = false;
        private string randomlyTimedDestChangeString;
        private bool followSingleBird = false;
        private string followSingleBirdString = "No";
        private bool pauseOn = false;
        private bool cohesionOn = true;
        private bool separationOn = true;
        private bool alignmentOn = true;
        private bool obstacleOn = true;
        private bool predatorOn = true;
```

```
        private string cohesionString;
        private string separationString;
        private string alignmentString;
        private string obstacleString;
        private string predatorString;
        private string pauseString;
        private float cohesionWeight = 1f;
        private float separationWeight = 12f;
        private float alignmentWeight = 1f;
        private float obstacleAvoidWeight = 13f;
        private float predatorAvoidWeight = 10f;

#region Accessors and Mutators

        public FlockBird[] FlockArray
        {
            get { return flockArray; }
            set { flockArray = value; }
        }

        public Obstacle[] ObstacleArray
        {
            get { return obstacleArray; }
            set { obstacleArray = value; }
        }

        public ArrayList ObstacleArrayList
        {
            get { return obstacleArrayList; }
            set
            {
                if (NumberOfObstacles < 10)
                    obstacleArrayList = value;
            }
        }

        public ArrayList FlockBirdArrayList
        {
            get { return flockBirdArrayList; }
            set
            {
                if (NumberOfBirds > 1 && NumberOfBirds < 80)
                flockBirdArrayList = value;
            }
        }

        public PredatorBird Predator
        {
            get { return predator; }
            set { predator = value; }
        }

        public int NumberOfBirds
        {
            get { return numberOfBirds; }
            set { numberOfBirds = value; }
        }

        public int NumberOfObstacles
        {
            get { return numberOfObstacles; }
            set { numberOfObstacles = value; }
        }
```

```
public bool CohesionOn
{
    get { return cohesionOn; }
    set { cohesionOn = value; }
}

public bool SeparationOn
{
    get { return separationOn; }
    set { separationOn = value; }
}

public bool AlignmentOn
{
    get { return alignmentOn; }
    set { alignmentOn = value; }
}

public bool ObstacleOn
{
    get { return obstacleOn; }
    set { obstacleOn = value; }
}

public bool PredatorOn
{
    get { return predatorOn; }
    set { predatorOn = value; }
}

public float DisplayWidth
{
    get { return displayWidth; }
    set { displayWidth = value; }
}

public float DisplayHeight
{
    get { return displayHeight; }
    set { displayHeight = value; }
}

public float CohesionWeight
{
    get { return cohesionWeight; }
    set { cohesionWeight = value; }
}

public float SeparationWeight
{
    get { return separationWeight; }
    set { separationWeight = value; }
}

public float AlignmentWeight
{
    get { return alignmentWeight; }
    set { alignmentWeight = value; }
}

public float ObstacleAvoidWeight
{
    get { return obstacleAvoidWeight; }
    set { obstacleAvoidWeight = value; }
```

```
        }

        public float PredatorAvoidWeight
        {
            get { return predatorAvoidWeight; }
            set { predatorAvoidWeight = value; }
        }

        public float SpeedRestricter
        {
            get { return speedRestricter; }
            set
            {
                speedRestricter = value;
                if (speedRestricter <= 0)
                    speedRestricter = 0.05f;
                if (speedRestricter > 2)
                    speedRestricter = 2f;
            }
        }

#endregion

        /// <summary>
        /// Constructor for Game1 object (the main simulation)
        /// </summary>
        public Game1()
        {
            //Allows control of the graphics device
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

            //Default window size 800x600 - this enlarges it
            graphics.PreferredBackBufferHeight = 700;
            graphics.PreferredBackBufferWidth = 1050;

            //Creates a new instance of UserTarget which the user
            //then controls to place destination co-ordinates
            birdTarget = new UserTarget(this);
            Components.Add(birdTarget);

            //Allows recognition of mouse position and button clicks
            Mouse.WindowHandle = Window.Handle;

            //Allows retrieval of keystrokes from keyboard
            currentKeyboardState = Keyboard.GetState();

            //Makes mouse pointer visible on game window
            this.IsMouseVisible = true;
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs before
        /// starting to run.
        /// This is where it can query for any required services and load
        /// any non-graphic related content.
        /// Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            displayWidth = graphics.GraphicsDevice.Viewport.Width;
            displayHeight = graphics.GraphicsDevice.Viewport.Height;
            flockArray = new FlockBird[numberOfBirds];
```

```
            obstacleArray = new Obstacle[numberOfObstacles];
            predator = new PredatorBird(this);

            for (int i = 0; i < NumberOfBirds; i++)
                FlockBirdArrayList.Add(new FlockBird(this));

            for (int i = 0; i < NumberOfObstacles; i++)
                ObstacleArrayList.Add(new Obstacle());

            Window.Title = "'Burds' Simulation by Robyn Backhouse";

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place
        /// to load all of the content.
        /// Create a new SpriteBatch for each sprite type,
        /// which can be used to draw textures.
        /// </summary>
        protected override void LoadContent()
        {
            fontBatch = new SpriteBatch(GraphicsDevice);
            font = Content.Load<SpriteFont>("Arial");
            fontB = Content.Load<SpriteFont>("ArialB");

            birdBatch = new SpriteBatch(GraphicsDevice);

            foreach (FlockBird bird in FlockBirdArrayList)
            {
                if (bird == FlockBirdArrayList[0])
                    bird.LoadContent(this.Content, leaderAssetName,
                     this);
                else
                    bird.LoadContent(this.Content, flockBirdAssetName,
                     this);
            }
            flockBirdArrayObject = FlockBirdArrayList.ToArray();
            FlockArray = new FlockBird[flockBirdArrayObject.Length];
            for (int i = 0; i < flockBirdArrayObject.Length; i++)
                flockArray[i] = (FlockBird)flockBirdArrayObject[i];

            predator.LoadContent(this.Content, predatorAssetName, this);

            obstacleBatch = new SpriteBatch(GraphicsDevice);
            foreach (Obstacle obstacle in ObstacleArrayList)
                obstacle.LoadContent(this.Content, obstacleAssetName);

            obstacleArrayObject = ObstacleArrayList.ToArray();
            obstacleArray = new Obstacle[obstacleArrayObject.Length];
            for (int i = 0; i < obstacleArrayObject.Length; i++)
                obstacleArray[i] = (Obstacle)obstacleArrayObject[i];

            for (int i = 0; i < obstacleArray.Length; i++)
                obstacleArray[i].PositionObstacle((i + 1), this);
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input.
        /// Update is called 60 times per second
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing
        ///  values.</param>
```

```
/// <if>If all birds have own destination</if>
/// <Else>Else birds all follow a leader still following
///  rules</Else>
protected override void Update(GameTime gameTime)
{
    if (!pauseOn)
    {
        if (!followSingleBird)
            foreach (FlockBird bird in flockArray)
                bird.UpdateBirdIndividuals(this);
        else
        {
            foreach (FlockBird bird in flockArray)
                if (bird == FlockArray[0])
                    bird.UpdateBirdLeader(gameTime, this);
                else
                    bird.UpdateBirdFollower(gameTime, this);
        }

        predator.UpdatePredator(this);

        if (randomlyTimedDestChange)
            RandomDestChange();
    }

    timePlayed++;

    AddUserPlaceObstacle();
    PlaceUserTarget();
    UpdateUserInput();
    UpdateGuiStrings();

    base.Update(gameTime);
}

/// <summary>
/// Randomly changes destination of all birds if option selected.
/// Randomly selects 2 numbers between zero and 300 and if they
/// are the same the all birds are given a new destination.
/// </summary>
private void RandomDestChange()
{
    int numOne = (int)RandomNumberGenerator.Next(300);
    int numTwo = (int)RandomNumberGenerator.Next(300);
    if (numOne == numTwo)
        foreach (FlockBird bird in flockArray)
            bird.NewDestination(this);
}

/// <summary>
/// Checks to see if user has placed an obstacle and
/// adds it in appropriate place if they have.
/// Updates Obstacle array.
/// </summary>
private void AddUserPlaceObstacle()
{
    MouseState obstaclePlacement = Mouse.GetState();
    if ((obstaclePlacement.LeftButton == ButtonState.Pressed) &&
       (NumberOfObstacles < 10) && (timePlayed < 500 ||
       timePlayed > 520))
    {
        NumberOfObstacles++;
        timePlayed = 500;
```

```
            ObstacleArrayList.Add(new Obstacle(obstaclePlacement.X -
                20, obstaclePlacement.Y - 20));
            obstacleArrayObject = ObstacleArrayList.ToArray();
            obstacleArray = new Obstacle[obstacleArrayObject.Length];
            for (int i = 0; i < obstacleArrayObject.Length; i++)
                obstacleArray[i] = (Obstacle)obstacleArrayObject[i];

            obstacleBatch = new SpriteBatch(GraphicsDevice);
            foreach (Obstacle obstacle in ObstacleArrayList)
                obstacle.LoadContent(this.Content,
                  obstacleAssetName);
        }
    }

    /// <summary>
    /// User can add more birds to flock in groups of 10.
    /// </summary>
    private void AddMoreBirds()
    {
        int numberOfBirdsAdded = 10;

        for (int i = 0; i < numberOfBirdsAdded; i++)
            FlockBirdArrayList.Add(new FlockBird(this));

        flockBirdArrayObject = FlockBirdArrayList.ToArray();
        flockArray = new FlockBird[flockBirdArrayObject.Length];
        for (int j = 0; j < flockBirdArrayObject.Length; j++)
            flockArray[j] = (FlockBird)flockBirdArrayObject[j];
        foreach (FlockBird bird in FlockArray)
            bird.LoadContent(this.Content, flockBirdAssetName, this);
        NumberOfBirds = FlockArray.Length;
    }

    /// <summary>
    /// Removes a bird from the flock when caught by predator
    /// </summary>
    /// <param name="j"></param>
    public void EatBird(int j)
    {
        FlockBirdArrayList.RemoveAt(j);
        flockBirdArrayObject = FlockBirdArrayList.ToArray();
        flockArray = new FlockBird[flockBirdArrayObject.Length];
        for (int i = 0; i < flockBirdArrayObject.Length; i++)
            flockArray[i] = (FlockBird)flockBirdArrayObject[i];
        NumberOfBirds = FlockArray.Length;
    }

    /// <summary>
    /// Checks whether user has placed a target destination for
    ///  all birds and if so, updates destinations of all birds
    ///  accordingly
    /// </summary>
    private void PlaceUserTarget()
    {
        MouseState userDirection = Mouse.GetState();
        if (userDirection.RightButton == ButtonState.Pressed)
        {
            Vector2 whereTo = new Vector2(userDirection.X,
                userDirection.Y);
            foreach (FlockBird bird in flockArray)
            {
                bird.UserSetDestination(this, whereTo);
                birdTarget.TargetPosition = whereTo;
            }
```

```
        }
    }

    /// <summary>
    /// Updates variables in response to user input on keyboard
    /// Effectively turns the steering rules on and off
    /// </summary>
    private void UpdateUserInput()
    {
        KeyboardState newState = Keyboard.GetState();

        // Key C controls the Cohesion rule
        if (newState.IsKeyDown(Keys.C))
            if (!currentKeyboardState.IsKeyDown(Keys.C))
                CohesionOn = !CohesionOn;

        // Key S controls the Separation rule
        if (newState.IsKeyDown(Keys.S))
            if (!currentKeyboardState.IsKeyDown(Keys.S))
                SeparationOn = !SeparationOn;

        // Key A controls the Alignment rule
        if (newState.IsKeyDown(Keys.A))
            if (!currentKeyboardState.IsKeyDown(Keys.A))
                AlignmentOn = !AlignmentOn;

        //Key O controls the Obstacle avoidance rule
        if (newState.IsKeyDown(Keys.O))
            if (!currentKeyboardState.IsKeyDown(Keys.O))
                ObstacleOn = !ObstacleOn;

        // Key R controls randomly timed destination change
        if (newState.IsKeyDown(Keys.R))
            if (!currentKeyboardState.IsKeyDown(Keys.R))
                randomlyTimedDestChange = !randomlyTimedDestChange;

        //Key F to swap between birds having individual
        //destinations or following a leader bird
        if (newState.IsKeyDown(Keys.F))
            if (!currentKeyboardState.IsKeyDown(Keys.F))
                followSingleBird = !followSingleBird;

        //Key + (plus, on number pad) to increase speed of flight
        if (newState.IsKeyDown(Keys.Up))
            if (!currentKeyboardState.IsKeyDown(Keys.Up))
                SpeedRestricter = SpeedRestricter + 0.1f;

        //Key - (minus, on number pad) to decrease speed of flight
        if (newState.IsKeyDown(Keys.Down))
            if (!currentKeyboardState.IsKeyDown(Keys.Down))
                SpeedRestricter = SpeedRestricter - 0.1f;

        //Key H to pause and resume game
        if (newState.IsKeyDown(Keys.H))
            if (!currentKeyboardState.IsKeyDown(Keys.H))
                pauseOn = !pauseOn;

        //Key P to select / deselect Predator bird
        if (newState.IsKeyDown(Keys.P))
            if (!currentKeyboardState.IsKeyDown(Keys.P))
                PredatorOn = !PredatorOn;

        //Key B to add 10 more birds to flock
        if (newState.IsKeyDown(Keys.B))
```

```
            if (!currentKeyboardState.IsKeyDown(Keys.B) &&
                NumberOfBirds <= 50)
                AddMoreBirds();

        currentKeyboardState = newState;
    }

    /// <summary>
    /// Reads the boolean values for rule status (whether rules are
    /// on or off) and updates strings accordingly for on screen
    /// display
    /// </summary>
    public void UpdateGuiStrings()
    {
        if (CohesionOn == true)
            cohesionString = "On";
        else cohesionString = "Off";

        if (SeparationOn == true)
            separationString = "On";
        else separationString = "Off";

        if (AlignmentOn == true)
            alignmentString = "On";
        else alignmentString = "Off";

        if (ObstacleOn == true)
            obstacleString = "On";
        else obstacleString = "Off";

        if (randomlyTimedDestChange == true)
            randomlyTimedDestChangeString = "On";
        else randomlyTimedDestChangeString = "Off";

        if (followSingleBird == true)
            followSingleBirdString = "Yes";
        else followSingleBirdString = "No";

        if (PredatorOn == true)
            predatorString = "On";
        else predatorString = "Off";

        if (pauseOn == true)
            pauseString = "On";
        else pauseString = "Off";
    }

    /// <summary>
    /// This is called when the simulation should draw itself.
    /// Draws the simulation window, text and sprites on screen
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing
    /// values.</param>
    protected override void Draw(GameTime gameTime)
    {
        graphics.GraphicsDevice.Clear(Color.MintCream);
        int bottom = 20;
        int spacer = 20;
        int spacer2 = 15;
        int top = 10;
        int horizontalBirdPos = 800;
        int horizontalTextOffset = 350;
```

```
Vector2 headingLine = new Vector2(5, DisplayHeight - (bottom
   + (9 * spacer)));
Vector2 addBirdsLine = new Vector2(5, DisplayHeight - (bottom
   + (8 * spacer)));
Vector2 cohesionGuiLine = new Vector2(5, DisplayHeight -
   (bottom + (7 * spacer)));
Vector2 separationGuiLine = new Vector2(5, DisplayHeight -
   (bottom + (6 * spacer)));
Vector2 alignmentGuiLine = new Vector2(5, DisplayHeight -
   (bottom + (5 * spacer)));
Vector2 predatorGuiLine = new Vector2(5, DisplayHeight -
   (bottom + (4 * spacer)));
Vector2 obstacleGuiLine = new Vector2(5, DisplayHeight -
   (bottom + (3 * spacer)));
Vector2 followLeaderBirdLine = new Vector2(5, DisplayHeight -
   (bottom + (2 * spacer)));
Vector2 pauseGuiLine = new Vector2(5, DisplayHeight - (bottom
   + 1 * spacer));
Vector2 randomTimeDestLine = new Vector2(5, DisplayHeight -
   (bottom + (0 * spacer)));
Vector2 numBirdsLine = new Vector2(horizontalBirdPos + 100,
   DisplayHeight - bottom);

Vector2 ObstaclePlaceGUILine = new
   Vector2(horizontalTextOffset, DisplayHeight - (bottom + 2
   * spacer));
Vector2 mouseClickDest = new Vector2(horizontalTextOffset,
   DisplayHeight - (bottom + 1 * spacer));
Vector2 speedGuiLine = new Vector2(horizontalTextOffset,
   DisplayHeight - (bottom + (0 * spacer)));

Vector2 ruleOffsetLine = new Vector2(20, 0);

Vector2 bird1Pos = new Vector2(horizontalBirdPos, top +
   spacer2 * 0);
Vector2 bird2Pos = new Vector2(horizontalBirdPos, top +
   spacer2 * 1);
Vector2 bird3Pos = new Vector2(horizontalBirdPos, top +
   spacer2 * 2);
Vector2 bird4Pos = new Vector2(horizontalBirdPos, top +
   spacer2 * 3);
Vector2 bird5Pos = new Vector2(horizontalBirdPos, top +
   spacer2 * 4);
Vector2 bird1Dest = new Vector2(5, top + spacer2 * 0);
Vector2 bird2Dest = new Vector2(5, top + spacer2 * 1);
Vector2 bird3Dest = new Vector2(5, top + spacer2 * 2);
Vector2 bird4Dest = new Vector2(5, top + spacer2 * 3);
Vector2 bird5Dest = new Vector2(5, top + spacer2 * 4);

fontBatch.Begin();

fontBatch.DrawString(fontB, "User Controls and Steering
   Rules:", headingLine, Color.DarkSlateGray);
fontBatch.DrawString(fontB, "B", addBirdsLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "P", predatorGuiLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "C", cohesionGuiLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "S", separationGuiLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "A", alignmentGuiLine,
   Color.DarkSlateBlue);
```

```
fontBatch.DrawString(fontB, "O", obstacleGuiLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "F", followLeaderBirdLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "R", randomTimeDestLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(fontB, "H", pauseGuiLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(font, "Predator: " + predatorString,
   predatorGuiLine + ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Cohesion: " + cohesionString,
   cohesionGuiLine + ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Separation: " + separationString,
   separationGuiLine + ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Alignment: " + alignmentString,
   alignmentGuiLine + ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Obstacle avoidance: " +
   obstacleString, obstacleGuiLine + ruleOffsetLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(font, "Follow leader bird: " +
   followSingleBirdString, followLeaderBirdLine +
   ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Randomly timed direction change
   for all birds: " + randomlyTimedDestChangeString,
   randomTimeDestLine + ruleOffsetLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Hold (pause) simulation: " +
   pauseString, pauseGuiLine + ruleOffsetLine,
   Color.DarkSlateBlue);
fontBatch.DrawString(font, "Current speed: " +
   SpeedRestricter + " Press Up or Down arrows to alter
   speed", speedGuiLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Left Click to place Obstacle",
   ObstaclePlaceGUILine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Right Click to place Target
   Destination", mouseClickDest, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Number of Birds: " +
   NumberOfBirds, numBirdsLine, Color.DarkSlateBlue);
fontBatch.DrawString(font, "Add Birds to flock (max 60)",
   addBirdsLine + ruleOffsetLine, Color.DarkSlateBlue);

if (FlockArray.Length > 0)
{
    fontBatch.DrawString(font, "Bird 1 Destination: " +
        FlockArray[0].BirdDestination, bird1Dest,
        Color.SlateBlue);
    fontBatch.DrawString(font, "Bird 1 Position: " +
        FlockArray[0].BirdPosition, bird1Pos,
        Color.SlateBlue);
}
if (FlockArray.Length > 1)
{
    fontBatch.DrawString(font, "Bird 2 Destination: " +
        FlockArray[1].BirdDestination, bird2Dest,
        Color.SlateBlue);
    fontBatch.DrawString(font, "Bird 2 Position: " +
        FlockArray[1].BirdPosition, bird2Pos,
        Color.SlateBlue);
}
if (FlockArray.Length > 2)
{
    fontBatch.DrawString(font, "Bird 3 Destination: " +
        FlockArray[2].BirdDestination, bird3Dest,
        Color.SlateBlue);
```

```
                fontBatch.DrawString(font, "Bird 3 Position: " +
                    FlockArray[2].BirdPosition, bird3Pos,
                    Color.SlateBlue);
            }
            if (FlockArray.Length > 3)
            {
                fontBatch.DrawString(font, "Bird 4 Destination: " +
                    FlockArray[3].BirdDestination, bird4Dest,
                    Color.SlateBlue);
                fontBatch.DrawString(font, "Bird 4 Position: " +
                    FlockArray[3].BirdPosition, bird4Pos,
                    Color.SlateBlue);
            }
            if (FlockArray.Length > 4)
            {
                fontBatch.DrawString(font, "Bird 5 Destination: " +
                    FlockArray[4].BirdDestination, bird5Dest,
                    Color.SlateBlue);
                fontBatch.DrawString(font, "Bird 5 Position: " +
                    FlockArray[4].BirdPosition, bird5Pos,
                    Color.SlateBlue);
            }
            fontBatch.End();

            birdBatch.Begin(SpriteBlendMode.AlphaBlend);
            foreach (FlockBird bird in flockArray)
                bird.Draw(this.birdBatch, this);
            if (PredatorOn)
                predator.Draw(this.birdBatch, this);
            birdBatch.End();

            obstacleBatch.Begin();
            foreach (Obstacle obstacle in obstacleArray)
                obstacle.Draw(this.obstacleBatch);
            obstacleBatch.End();

            base.Draw(gameTime);
        }
    }
}
```

D.3 Class: Bird (Base Class)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    public class Bird
    {
        //current position of bird
        private Vector2 birdPosition = new Vector2();
        protected Vector2 birdDestination = new Vector2(850f, 300f);
        //How close birds get to centre of obstacle
        private float obstacleAvoidanceDistance = 40;
        //texture to use when drawing bird
        private Texture2D birdTexture;
        private float birdTextureHeight;    //Height of bird picture
        private float birdTextureWidth;     //Width of bird picture
        //Used to control bird speed
        private static float speedLimiter = 0.015f;
        //Size of neighbourhood radius
        private float neighbourDistance = 200;

        // Max and min x and y values for edges of screen for birds
        private float minX;         // 0 minus width of bird
        private float minY;         // 0 minus height of bird
        private float maxX;         // Width of window
        private float maxY;         // Height of window

      #region Constructors

        /// <summary>
        /// Bird object constructor used for simulation.
        /// Gives the bird co-ordinates and a destination
        /// </summary>
        public Bird(Game1 theGame)
        {
            this.PlaceNewBird();
            this.NewDestination(theGame);
        }

        /// <summary>
        /// Bird object constructor for making bird with specific
        /// co-ordinates.
        /// Used mainly during testing procedures rather than for
        /// simulation.
        /// </summary>
        public Bird(float posx, float posy, float destx, float desty)
        {
            birdPosition.X = posx;
            birdPosition.Y = posy;
            birdDestination.X = destx;
            birdDestination.Y = desty;
        }
```

```
#endregion

#region Accessors and Mutators

  public Vector2 BirdPosition
  {
      get { return birdPosition; }
      set { birdPosition = value; }
  }

  public Vector2 BirdDestination
  {
      get { return birdDestination; }
      set { birdDestination = value; }
  }

  public float BirdTextureHeight
  {
      get { return birdTextureHeight; }
      set { birdTextureHeight = value; }
  }

  public float BirdTextureWidth
  {
      get { return birdTextureWidth; }
      set { birdTextureWidth = value; }
  }

  public float MinX
  {
      get { return minX; }
      set { minX = value; }
  }

  public float MinY
  {
      get { return minY; }
      set { minY = value; }
  }

  public float MaxX
  {
      get { return maxX; }
      set { maxX = value; }
  }

  public float MaxY
  {
      get { return maxY; }
      set { maxY = value; }
  }

  public static float SpeedLimiter
  {
      get { return Bird.speedLimiter; }
  }

  public float NeighbourDistance
  {
      get { return neighbourDistance; }
      set { neighbourDistance = value; }
  }

#endregion
```

```
/// <summary>
/// Called by XNA framework to load content
/// </summary>
public void LoadContent(ContentManager theContentManager, string
   theAssetName, Game1 theGame)
{
    birdTexture =
       theContentManager.Load<Texture2D>(theAssetName);
    BirdTextureHeight = birdTexture.Height;
    BirdTextureWidth = birdTexture.Width;
    MaxX = theGame.DisplayWidth;
    MaxY = theGame.DisplayHeight;
    MinX = birdTexture.Width * -1;
    MinY = birdTexture.Height * -1;
}

/// <summary>
/// Cohesion rule: birds fly towards other birds,
/// or centre of flock
/// </summary>
protected Vector2 FlockCohesion(Game1 theGame, Bird mybird)
{
    Vector2 flockCentre = new Vector2(0, 0);
    int numberOfBirdsInNeighbourhood = 0;

    foreach (Bird bird in theGame.FlockArray)
    {
        if ((mybird != bird) && (mybird.EuclideanDistance(bird) <
          neighbourDistance) && (mybird.AngleBetweenBirds(bird) <
          140 && mybird.AngleBetweenBirds(bird) > -140) )
        {
            flockCentre = (flockCentre + bird.BirdPosition);
            numberOfBirdsInNeighbourhood++;
        }
    }
    if (numberOfBirdsInNeighbourhood > 0)
        return ((flockCentre / numberOfBirdsInNeighbourhood) -
          mybird.BirdPosition) * theGame.CohesionWeight;
    else return flockCentre;
}

/// <summary>
/// Obstacle avoidance rule - birds avoid hitting obstacles
/// </summary>
protected Vector2 ObstacleAvoidance(Game1 theGame, Bird mybird)
{
    Vector2 avoidObstacle = new Vector2(0, 0);

    foreach (Obstacle obstacle in theGame.ObstacleArray)
    {
        if (mybird.EuclideanDistanceObs(obstacle) <
          ((obstacle.ObstacleTextureWidth / 2) +
          obstacleAvoidanceDistance))
        {
            avoidObstacle.X = avoidObstacle.X -
               ((obstacle.ObstaclePosition.X +
               (obstacle.ObstacleTextureWidth / 2)) -
               mybird.BirdPosition.X);
            avoidObstacle.Y = avoidObstacle.Y -
               ((obstacle.ObstaclePosition.Y +
               (obstacle.ObstacleTextureHeight / 2)) -
               mybird.BirdPosition.Y);
        }
```

```
    }
    return avoidObstacle * theGame.ObstacleAvoidWeight;
}

/// <summary>
/// Calculates the Euclidean distance between 2 birds
/// This is always a positive number and all calculated values
/// are therefore positive
/// </summary>
public float EuclideanDistance(Bird bird2)
{
    float x = (float)Math.Pow((bird2.BirdPosition.X -
       this.BirdPosition.X), 2);
    float y = (float)Math.Pow((bird2.BirdPosition.Y -
       this.BirdPosition.Y), 2);
    return (float)Math.Sqrt(x + y);
}

/// <summary>
/// Calculates the Euclidean distance between an obstacle and
/// a bird
/// This is always a positive number, and all calculated values
/// are therefore positive
/// </summary>
public float EuclideanDistanceObs(Obstacle obstacle)
{
    float x = (float)Math.Pow(((obstacle.ObstaclePosition.X +
       (obstacle.ObstacleTextureWidth / 2)) -
       this.BirdPosition.X), 2);
    float y = (float)Math.Pow(((obstacle.ObstaclePosition.Y +
       (obstacle.ObstacleTextureHeight / 2)) -
       this.BirdPosition.Y), 2);
    return (float)Math.Sqrt(x + y);
}

/// <summary>
/// Calculates Euclidean distance between bird position and
/// its destination
/// Used for direction/rotation facing of bird sprite
/// </summary>
public double EuclideanDistanceVectors(Vector2
   combinedDestination)
{
    float x = (float)Math.Pow((combinedDestination.X -
       this.BirdPosition.X), 2);
    float y = (float)Math.Pow((combinedDestination.Y -
       this.BirdPosition.Y), 2);
    return (double)Math.Sqrt(x + y);
}

/// <summary>
/// Returns the angle between the heading of one bird and
/// the position of the second bird
/// </summary>
/// <param name="bird2Dest">comparing a bird "this" to a second
/// bird </param>
/// <returns>the angle in degrees</returns>
public int AngleBetweenBirds(Bird bird2)
{
    //To ensure don't divide by zero
    if (this.BirdDestination.X == 0)
        this.birdDestination.X = 0.1f;
```

```
        double tanBird1 = this.BirdDestination.Y /
          this.BirdDestination.X;

        double mybirdFacing = Math.Atan(tanBird1);

        Vector2 positionDifference = new Vector2(bird2.BirdPosition.X
          - this.BirdPosition.X, bird2.BirdPosition.Y -
          this.BirdPosition.Y);

        //To ensure don't divide by zero
        if (positionDifference.X == 0)
            positionDifference.X = 0.1f;

        double tan = positionDifference.Y / positionDifference.X;
        double radiansBird2 = Math.Atan(tan);

        double bird1Angle = mybirdFacing * 180 / Math.PI;
        double bird2Angle = radiansBird2 * 180 / Math.PI;
        double answer = bird1Angle - bird2Angle;

        return (int)answer;
    }

    /// <summary>
    /// compares the angle between a bird and a stationary obstacle.
    /// </summary>
    /// <returns>angle in degrees</returns>
    public int AngleBetweenBirdObstacle(Obstacle obstacle)
    {
        //To ensure don't divide by zero
        if (this.BirdDestination.X == 0)
            this.birdDestination.X = 0.1f;

        double tanBird = this.BirdDestination.Y /
           this.BirdDestination.X;

        double birdFacing = Math.Atan(tanBird);

        Vector2 obstaclePos = new
           Vector2(obstacle.ObstaclePosition.X,
           obstacle.ObstaclePosition.Y);
        Vector2 positionDifference = new
           Vector2(obstacle.ObstaclePosition.X - this.BirdPosition.X,
           obstacle.ObstaclePosition.Y - this.BirdPosition.Y);

        //To ensure don't divide by zero
        if (positionDifference.X == 0)
            positionDifference.X = 0.1f;

        double tan = positionDifference.Y / positionDifference.X;
        double radiansObstacle = Math.Atan(tan);

        double birdAngle = birdFacing * 180 / Math.PI;
        double obstacleAngle = radiansObstacle * 180 / Math.PI;
        double answer = birdAngle - obstacleAngle;

        return (int)answer;
    }

    /// <summary>
    /// Calculates rotation angle of bird / direction of travel as
    /// an angle in radians
    /// </summary>
    public double AngleBirdFacing()
```

```
{
    //To ensure don't divide by zero
    if (this.BirdDestination.X == 0)
        this.birdDestination.X = 0.1f;

    double tan = this.BirdDestination.Y / this.BirdDestination.X;

    //rotate 90 degrees to correct rotation
    double radians = Math.Atan(tan) - 1.57079633;

    // If X is positive, add 180 degrees in radians
    if (this.BirdDestination.X > 0)
        radians = radians + (2 * 1.57079633);
    return radians;
}

/// <summary>
/// Vector maths sometimes gives birds co-ordinates of up to
/// 5000, which is way off the screen.
/// This procedure normalises them back to a reasonable
/// figure, which keeps their movement and speed more constant
/// and controlled. Speed is relative to distance-to-destination
/// </summary>
public void BirdDistanceController(Game1 theGame)
{
    float extraBit = 200;
    if (this.BirdDestination.X > (2 * theGame.DisplayWidth))
        this.birdDestination.X = ((BirdDestination.X / 2) +
            extraBit);
    if (this.BirdDestination.X < (2 * theGame.DisplayWidth) * -1)
        this.birdDestination.X = ((BirdDestination.X / 2) -
            extraBit);
    if (this.BirdDestination.Y > (2 * theGame.DisplayHeight))
        this.birdDestination.Y = ((BirdDestination.Y / 2) +
            extraBit);
    if (this.BirdDestination.Y < (2 * theGame.DisplayHeight * -
            1))
        this.birdDestination.Y = ((BirdDestination.Y / 2) -
            extraBit);
}

/// <summary>
/// Places a new bird within a set square on the screen.
/// Can change where this square appears, or the size of it.
/// </summary>
public void PlaceNewBird()
{
    Vector2 returnVector = new Vector2(0, 0);
    float xOffset = 20;
    float yOffset = 150;
    int squareSize = 100;

    returnVector.X = (RandomNumberGenerator.Next(squareSize) +
       xOffset);
    returnVector.Y = (RandomNumberGenerator.Next(squareSize) +
       yOffset);
    BirdPosition = returnVector;
}

/// <summary>
/// Sets randomly chosen coordinates for birds destination.
/// </summary>
public void NewDestination(Game1 theGame)
{
```

```
        // Add 0.5f to force correct rounding when converting
        // to integer
        float y =
           RandomNumberGenerator.Next(Convert.ToInt32(theGame.Display
           Height + 0.5f));
        float x =
           RandomNumberGenerator.Next(Convert.ToInt32(theGame.Display
           Width + 0.5f));

        // Chance for coordinates to be positive or negative on new
        // direction.
        int vertDirection = (int)RandomNumberGenerator.Next(2);
        int horizontDirection = (int)RandomNumberGenerator.Next(2);

        if (vertDirection == 1)
            y = y * -1;
        if (horizontDirection == 1)
            x = x * -1;

        birdDestination.Y = y;
        birdDestination.X = x;
    }

    /// <summary>
    /// When bird goes off edge of screen, wraps to other side
    /// (like globe/map of world)
    /// </summary>
    public void WrapBird(Game1 theGame)
    {
        if (BirdPosition.X > theGame.DisplayWidth)
            birdPosition.X = 0 - this.BirdTextureWidth;
        if (BirdPosition.X < (0 - this.BirdTextureWidth))
            birdPosition.X = theGame.DisplayWidth;

        if (BirdPosition.Y > theGame.DisplayHeight)
            birdPosition.Y = 0 - this.BirdTextureHeight;
        if (BirdPosition.Y < (0 - this.BirdTextureHeight))
            birdPosition.Y = theGame.DisplayHeight;
    }

    /// <summary>
    /// Draws the bird – both predator and flockbirds
    /// </summary>
    public void Draw(SpriteBatch birdBatch, Game1 theGame)
    {
        birdBatch.Draw
        (
            birdTexture,    //the Bird texture
            birdPosition,   //the Bird position

            //source rectangle of bird sprite - used for rotation
            new Rectangle(0, 0, (int)this.BirdTextureWidth,
                (int)this.BirdTextureHeight),
            Color.White,    //Overlay colour - white for no colour
                overlay
            (float)this.AngleBirdFacing(),  //Rotation angle for Bird
                sprite in radians

            //Origin - middle of the Bird sprite texture - rotates
                around its centre.
            new Vector2(this.BirdTextureWidth / 2,
                this.BirdTextureHeight / 2),
            1.0f,   //float Scale - Not used. used for 2D
                representatin of 3D world
```

```
                SpriteEffects.None,  //Sprite effects - Not used, but
                    must be in definition
                1.0f    //float Layer depth - not used, but must be
                    present. Default to 1.0
            );
        }
    }
}
```

D.4 Class: Flock Bird (Inherited Class)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    public class FlockBird : Bird
    {
        //How close birds get to other birds
        private float collisionDistance = 25;
        //Minimum distance from Predator
        private float predatorAvoidanceDistance = 100;

        /// <summary>
        /// Constructor for FlockBird class.  Used for Burds Simulation
        /// </summary>
        /// <param name="theGame"></param>
        public FlockBird(Game1 theGame)
            : base(theGame)
        {    }

        /// <summary>
        /// Constructor for FlockBird class.
        /// Used for test suite to set up a custom test flock
        /// </summary>
        public FlockBird( float posx, float posy, float destx, float
            desty)
            : base (posx, posy, destx, desty)
        {    }


        /// <summary>
        /// Calculates a headingVector from the steering rules which is
        /// combined with bird's destination.
        /// </summary>
        public Vector2 CalculateCorrectedHeadingAllBirds(Game1 theGame)
        {
            Vector2 headingVector = new Vector2(0, 0);

            if (theGame.PredatorOn)
            {
                if (this.EuclideanDistance(theGame.Predator) <
                    predatorAvoidanceDistance)
                {
                    if (theGame.ObstacleOn)
                    {
                        Vector2 obstacleAvoid =
                            this.ObstacleAvoidance(theGame, this);
                        headingVector = headingVector + obstacleAvoid;
                    }

                    if (theGame.SeparationOn)
                    {
```

```
                    Vector2 separation =
                        this.FlockSeparation(theGame, this);
                    headingVector = headingVector + separation;
                }

                Vector2 predatoravoid =
                        this.FlockPredatorAvoid(theGame, this);
                headingVector = headingVector + predatoravoid;
                return headingVector;
            }
        }

        if (theGame.CohesionOn)
        {
            Vector2 cohesion = this.FlockCohesion(theGame, this);
            if (theGame.PredatorOn &&
                this.EuclideanDistance(theGame.Predator) <
                predatorAvoidanceDistance + 100)
                ;   //headingVector = headingVector
            else headingVector = headingVector + cohesion;
        }
        if (theGame.SeparationOn)
        {
            Vector2 separation = this.FlockSeparation(theGame, this);
            headingVector = headingVector + separation;
        }
        if (theGame.AlignmentOn)
        {
            Vector2 alignment = this.FlockAlignment(theGame, this);
            if (theGame.PredatorOn &&
                this.EuclideanDistance(theGame.Predator) <
                predatorAvoidanceDistance + 100)
                ;   //headingVector = headingVector
            headingVector = headingVector + alignment;
        }

        if (theGame.ObstacleOn)
        {
            Vector2 obstacleAvoid = this.ObstacleAvoidance(theGame,
                this);
            headingVector = headingVector + obstacleAvoid;
        }

        return headingVector;
    }

    /// <summary>
    /// Calculates a headingVector from the steering rules for the
    /// Leader Bird which is combined with bird's destination.
    /// Reduced rules - only obstacle avoidance.
    /// The leader knows exactly where it is going, regardless of
    /// other birds and does not attempt to separate from them
    /// as it will be in the lead.
    /// </summary>
    public Vector2 CalculateCorrectedHeadingLeaderBird(Game1 theGame)
    {
        Vector2 headingVector = new Vector2(0, 0);

        if (theGame.PredatorOn &&
           (this.EuclideanDistance(theGame.Predator) <
           predatorAvoidanceDistance))
        {
            if (theGame.ObstacleOn)
            {
```

```
                Vector2 obstacleAvoid =
                  this.ObstacleAvoidance(theGame, this);
                headingVector = headingVector + obstacleAvoid;
            }

            if (theGame.SeparationOn)
            {
                Vector2 separation = this.FlockSeparation(theGame,
                  this);
                headingVector = headingVector + separation;
            }

            Vector2 predatoravoid = this.FlockPredatorAvoid(theGame,
              this);
            headingVector = headingVector + predatoravoid;
            return headingVector;
        }

        if (theGame.SeparationOn)
        {
            Vector2 separation = this.FlockSeparation(theGame, this);
            headingVector = headingVector + separation;
        }

        if (theGame.ObstacleOn)
        {
            Vector2 obstacleAvoid = this.ObstacleAvoidance(theGame,
              this);
            headingVector = headingVector + obstacleAvoid;
        }

        return headingVector;
    }

    /// <summary>
    /// Separation rule: birds avoid collisions with other birds
    /// and maintain a set distance from other birds
    /// </summary>
    private Vector2 FlockSeparation(Game1 theGame, Bird mybird)
    {
        Vector2 separationVector = new Vector2(0, 0);

        foreach (Bird bird in theGame.FlockArray)
        {
            if ((mybird != bird) && (mybird.EuclideanDistance(bird) <
              collisionDistance) && ((mybird.AngleBetweenBirds(bird)
              < 140) && (mybird.AngleBetweenBirds(bird) > -140)))

                separationVector = separationVector -
                  (bird.BirdPosition - mybird.BirdPosition);
        }
        return separationVector * theGame.SeparationWeight;
    }

    /// <summary>
    /// Alignment rule: birds align their heading with flockmates
    /// </summary>
    private Vector2 FlockAlignment(Game1 theGame, Bird mybird)
    {
        Vector2 alignmentVector = new Vector2();
        alignmentVector = mybird.BirdDestination;
        int numberOfBirdsInNeighbourhood = 1;

        foreach (Bird bird in theGame.FlockArray)
```

```
        {
            if ((mybird != bird) && (mybird.EuclideanDistance(bird) <
                NeighbourDistance) && (mybird.AngleBetweenBirds(bird)
                < 140 && mybird.AngleBetweenBirds(bird) > -140))
            {
                alignmentVector = alignmentVector +
                    bird.BirdDestination;
                numberOfBirdsInNeighbourhood++;
            }
        }

        alignmentVector = alignmentVector /
           numberOfBirdsInNeighbourhood;

        //calculate percieved destination, then add about an eighth
        //of this to the birds current destination.
        return ((alignmentVector - mybird.BirdDestination) / 8) *
           theGame.AlignmentWeight;
    }

    /// <summary>
    /// Steering rule for predator avoidance.
    /// </summary>
    private Vector2 FlockPredatorAvoid(Game1 theGame, Bird mybird)
    {
        Vector2 predatorAvoid = new Vector2(0, 0);

            predatorAvoid.X = predatorAvoid.X -
                (theGame.Predator.BirdPosition.X -
                mybird.BirdPosition.X);
            predatorAvoid.Y = predatorAvoid.Y -
                (theGame.Predator.BirdPosition.Y -
                mybird.BirdPosition.Y);

        return predatorAvoid * theGame.PredatorAvoidWeight;
    }

    /// <summary>
    /// Updates the bird current position to the new position - makes
    /// bird move.
    /// This is called when ALL birds have individual destinations.
    /// More like real birds..
    /// Move the bird by destination and steering rules, scaled by
    /// elapsed time and speedRestricter
    /// Changes destination coordinates and wraps bird when goes off
    /// edge of screen
    /// </summary>
    public void UpdateBirdIndividuals( Game1 theGame)
    {
        if (BirdPosition.X >= base.MaxX || BirdPosition.X <=
           base.MinX || BirdPosition.Y <= base.MinY || BirdPosition.Y
           >= base.MaxY)
        {
            WrapBird(theGame);
            NewDestination(theGame);
        }

        Vector2 headingVector =
           CalculateCorrectedHeadingAllBirds(theGame);
        BirdDestination = (BirdDestination + headingVector);
        this.BirdDistanceController(theGame);
        BirdPosition = BirdPosition + BirdDestination * SpeedLimiter
           * theGame.SpeedRestricter;
    }
```

```
        /// <summary>
        /// Updates the bird current position to the new position - makes
        /// bird move.
        /// This is called on all birds except the leader bird, when only
        /// leader bird has destination.
        /// Others only obey steering rules - do not have own
        /// destinations
        /// Move the bird by destination of leader bird, scaled by
        /// elapsed time and speedRestricter
        /// </summary>
        public void UpdateBirdFollower(GameTime gameTime, Game1 theGame)
        {
            if (BirdPosition.X >= MaxX || BirdPosition.X <= MinX ||
               BirdPosition.Y <= MinY || BirdPosition.Y >= MaxY)
                WrapBird(theGame);

            Vector2 headingVector =
               this.CalculateCorrectedHeadingAllBirds(theGame);
            BirdDestination = (theGame.FlockArray[0].BirdPosition -
               this.BirdPosition) + headingVector;
            this.BirdDistanceController(theGame);
            BirdPosition = BirdPosition + BirdDestination * SpeedLimiter
               * (theGame.SpeedRestricter * 3);
        }

        /// <summary>
        /// Updates the bird current position to the new position - makes
        /// bird move.
        /// This is for the leader bird only, so does not include
        /// cohesion or alignment
        /// Move the bird by destination and steering rules, scaled by
        /// elapsed time and speedRestricter.
        /// Changes destination coordinates and wraps bird when goes off
        /// edge of screen
        /// </summary>
        public void UpdateBirdLeader(GameTime gameTime, Game1 theGame)
        {
            if (BirdPosition.X >= MaxX || BirdPosition.X <= MinX ||
               BirdPosition.Y <= MinY || BirdPosition.Y >= MaxY)
            {
                WrapBird(theGame);
                NewDestination(theGame);
            }

            Vector2 headingVector =
               this.CalculateCorrectedHeadingLeaderBird(theGame);
            BirdDestination = BirdDestination + headingVector;
            this.BirdDistanceController(theGame);
            BirdPosition = BirdPosition + BirdDestination * (SpeedLimiter
               * 3) * (theGame.SpeedRestricter / 2);
        }

        /// <summary>
        /// Sets bird destination to the point the user clicked
        /// </summary>
        public void UserSetDestination(Game1 theGame, Vector2 userDest)
        {
            BirdDestination = userDest - BirdPosition;
        }
    }
}
```

D.5 Class: Predator Bird (Inherited Class)

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    public class PredatorBird : Bird
    {
        /// <summary>
        /// Constructor for Predator Bird.  Calls the base Bird class.
        /// </summary>
        /// <param name="theGame"></param>
        public PredatorBird(Game1 theGame)
            : base(theGame)
        {    }

        /// <summary>
        /// Constructor for Predator Bird, specifying position and
        /// destination co-ordinates
        /// </summary>
        public PredatorBird( float posx, float posy, float destx, float
             desty)
            : base (posx, posy, destx, desty)
        {    }


        /// <summary>
        /// Updates the position and destination of the predator.
        /// </summary>
        public void UpdatePredator(Game1 theGame)
        {
            if (BirdPosition.X >= MaxX || BirdPosition.X <= MinX ||
               BirdPosition.Y <= MinY || BirdPosition.Y >= MaxY)
            {
                WrapBird(theGame);
                NewDestination(theGame);
            }

            Vector2 headingVector =
               this.CalculateNewHeadingPredator(theGame);
            BirdDestination = BirdDestination + headingVector;
            this.BirdDistanceController(theGame);
            BirdPosition = BirdPosition + BirdDestination * (
               SpeedLimiter * 2.5f ) * (theGame.SpeedRestricter / 2);

            if (theGame.PredatorOn)
                for (int i = 0; i < theGame.FlockArray.Length; i++)
                {
                    if (this.EuclideanDistance(theGame.FlockArray[i])<10)
                        theGame.EatBird(i);
                }
        }

        /// <summary>
```

```
        /// Calculates a new destination for the predator, considering
        /// obstacles and flock birds or prey.
        /// </summary>
        /// <returns>vector for new destination</returns>
        public Vector2 CalculateNewHeadingPredator(Game1 theGame)
        {
            Vector2 headingVector = new Vector2(0, 0);

            if (theGame.ObstacleOn)
            {
                Vector2 obstacleAvoid = this.ObstacleAvoidance(theGame,
                    this);
                headingVector = headingVector + obstacleAvoid;
            }

            Vector2 cohesion = this.FlockCohesion(theGame, this);
            headingVector = headingVector + cohesion;

            return headingVector;
        }
    }
}
```

D.6 Class: Obstacle

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    public class Obstacle
    {
        Texture2D obstacleTexture;  // Texture to render for obstacles
        Vector2 obstaclePosition = new Vector2();
        private float obstacleTextureHeight;
        private float obstacleTextureWidth;
        Vector2 obstacleCentre = new Vector2();

        /// <summary>
        /// Constructor for Obstacle class taking no arguments
        /// </summary>
        public Obstacle()
        {
            obstaclePosition.X = obstaclePosition.Y = 0;
        }

        /// <summary>
        /// Constructor for Obstacle class taking co-ordinates
        /// used mainly for testing
        /// </summary>
        public Obstacle(float x, float y)
        {
            obstaclePosition.X = x;
            obstaclePosition.Y = y;
        }

        public Vector2 ObstaclePosition
        {
            get { return obstaclePosition; }
            set { obstaclePosition = value; }
        }

        public Vector2 ObstacleCentre
        {
            get { return obstacleCentre; }
            set { obstacleCentre = value; }
        }

        public void UpdateObstaclePosition(float x, float y)
        {
            obstaclePosition.X = x;
            obstaclePosition.Y = y;
        }

        public float ObstacleTextureHeight
        { get { return obstacleTextureHeight; } }

        public float ObstacleTextureWidth
```

```
        { get { return obstacleTextureWidth; } }

        /// <summary>
        /// Called by XNA framework to load content for this class
        /// </summary>
        public void LoadContent(ContentManager theContentManager, string
            theAssetName)
        {
            obstacleTexture =
                theContentManager.Load<Texture2D>(theAssetName);
            obstacleTextureHeight = obstacleTexture.Height;
            obstacleTextureWidth = obstacleTexture.Width;
            obstacleCentre.X = (ObstaclePosition.X +
                (obstacleTexture.Width / 2));
            obstacleCentre.Y = (ObstaclePosition.Y +
                (obstacleTexture.Height / 2));
        }

        /// <summary>
        /// Draws the obstacle sprites on the screen
        /// </summary>
        public void Draw(SpriteBatch obstacleBatch)
        {
            obstacleBatch.Draw(obstacleTexture, obstaclePosition,
                Color.White);
        }

        /// <summary>
        /// Positions hard coded obstacles on screen
        /// </summary>
        public void PositionObstacle(int i, Game1 theGame)
        {
            // Set obstacles in place depending upon number chosen
            if (i == 2)
                UpdateObstaclePosition((theGame.DisplayWidth / 6),
                    (theGame.DisplayHeight / 7) * 2);

            if (i == 1)
                UpdateObstaclePosition(((theGame.DisplayWidth / 4) *
                    3.2f), ((theGame.DisplayHeight -
                    ObstacleTextureHeight) / 2));
        }
    }
}
```

D.7 Class: User Target

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Burds2D
{
    /// <summary>
    /// This is a drawable game component
    /// </summary>
    internal class UserTarget :
         Microsoft.Xna.Framework.DrawableGameComponent
    {

        private Texture2D targetTexture;
        private Vector2 targetPosition;
        private Color targetOverlayColour;
        private float targetFader;
        private SpriteBatch targetBatch;

        internal Vector2 TargetPosition
        {
            get { return targetPosition; }
            set
            {   //Sets new target values every time user clicks mouse

                // Centres the target position
                value.X = value.X - (targetTexture.Width / 2);
                value.Y = value.Y - (targetTexture.Height / 2);
                targetPosition = value;
                // Sets the fading time for target sprite
                targetFader = 3f;
                // Makes the target visible - "Visible" is XNA component
                Visible = true;
            }
        }

        /// <summary>
        /// Constructor for Target class
        /// </summary>
        internal UserTarget(Game game)
            : base(game)
        {
            targetOverlayColour = Color.WhiteSmoke;
        }

        /// <summary>
        /// Allows the game component to perform any initialization it
        /// needs to before starting to run.
        /// This is where it can query for any required services and
        /// load content.
        /// </summary>
        public override void Initialize()
        {
            base.Initialize();
```

```
        }

        /// <summary>
        /// Allows the game component to load graphics content
        /// </summary>
        protected override void LoadContent()
        {
            targetTexture = Game.Content.Load<Texture2D>("userTarget");
            targetBatch = new SpriteBatch(GraphicsDevice);
            base.LoadContent();
        }

        /// <summary>
        /// Allows the game component to update itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing
        /// values.</param>
        public override void Update(GameTime gameTime)
        {
            if (Visible)
            {
                //Fades the picture of the user pointer incrementally
                //over 2 seconds
                targetFader = targetFader - 0.02f;

                //After 2 seconds, timer is zero and pointer no longer
                //visible
                if (targetFader <= 0)
                {
                    Visible = false;
                }
                else
                {
                    Vector4 colour = new
                      Vector4(targetOverlayColour.ToVector3(),
                      targetFader);
                    targetOverlayColour = new Color(colour);
                }
            }
            base.Update(gameTime);
        }

        /// <summary>
        /// Allows the game component to draw itself
        /// </summary>
        public override void Draw(GameTime gameTime)
        {
            targetBatch.Begin();
            targetBatch.Draw(targetTexture, targetPosition,
               targetOverlayColour);
            targetBatch.End();
            base.Draw(gameTime);
        }
    }
}
```

D.8 Class: Random Number Generator

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Burds2D
{
    /// <summary>
    /// Returns a non-negative random number less than the specified
    /// maximum(includes zero as a possible return value.
    /// </summary>

    public class RandomNumberGenerator
    {
        private static Random r;

        static RandomNumberGenerator()
        {
            r = new Random();
        }

        public static float Next(int range)
        {
            return r.Next(range);
        }
    }
}
```

APPENDIX E

Code for Test Suite:

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.GamerServices;
using Burds2D;

namespace Burds2D
{

    /// <summary>
    /// Summary description for UnitTest1
    /// </summary>
    [TestClass]
    public class UnitTest1
    {
        Game1 FlockingBirds = new Game1();
        private FlockBird[] testFlock;
        private PredatorBird testPredator;
        Bird one = new Bird(73, 189, 207, 643);
        Bird two = new Bird(48, 234, 243, -120);
        Bird three = new Bird(87, 202, 24, -338);
        Bird four = new Bird(57, 167, 529, 590);
        Bird bird1 = new Bird(-250, -400, 0, 0);
        Bird bird2 = new Bird(520, 100, 0, 0);
        Bird bird3 = new Bird(200, 300, 0, 0);
        Bird bird4 = new Bird(320, 200, 0, 0);
        Bird bird5 = new Bird(10, 300, 0, 0);
        Bird bird6 = new Bird(3567, 49345, 0, 0);
        Bird bird7 = new Bird(160, 300, 0, 0);
        Bird bird8 = new Bird(840, 398, 0, 0);
        Bird bird9 = new Bird(2736, 85367, 0, 0);
        Bird bird10 = new Bird(300, 400, 0, 0);
        Bird bird11 = new Bird(300, 400, 6000, 3000);
        Bird bird12 = new Bird(200, 300, -4000, -4398);
        Bird bird13 = new Bird(300, 200, 500, 400);
        Bird bird14 = new Bird(600, 400, -300, -200);
        Bird birdwrap = new Bird(2000, 1500, 19, 100);

        Obstacle obstacle = new Obstacle(642, 385);
        Obstacle obstacle1 = new Obstacle(45, 7643);
        Obstacle obstacle2 = new Obstacle(230f, 250f);

        public UnitTest1()
        {
            testFlock = new FlockBird[5];
            testPredator = new PredatorBird(300, 300, 800, 400);

            testFlock[0] = new FlockBird(400, 400, 600, 600);
            testFlock[1] = new FlockBird(500, 500, 700, 700);
            testFlock[2] = new FlockBird(100, 100, 300, 400);
            testFlock[3] = new FlockBird(200, 200, -100, -50);
            testFlock[4] = new FlockBird(600, 600, -300, -200);
        }

        private TestContext testContextInstance;
```

```
/// <summary>
///Gets or sets the test context which provides
///information about and functionality for the current test run.
///</summary>
public TestContext TestContext
{
    get
    {
        return testContextInstance;
    }
    set
    {
        testContextInstance = value;
    }
}

/// <summary>
/// Tests the setting up of the array of flock of birds.
/// </summary>
[TestMethod]
public void testFlockArraySetup()
{
    Assert.AreEqual(600, testFlock[0].BirdDestination.X);
    Assert.AreEqual(-200, testFlock[4].BirdDestination.Y);
    Assert.AreEqual(200, testFlock[3].BirdPosition.X);
    Assert.AreEqual(100, testFlock[2].BirdPosition.Y);
}

/// <summary>
/// Tests the WrapBird function
/// </summary>
[TestMethod]
public void WrapBirdTest()
{
    birdwrap.WrapBird(FlockingBirds);
    Assert.AreEqual(0, birdwrap.BirdPosition.X);
    Assert.AreEqual(0, birdwrap.BirdPosition.Y);
}

/// <summary>
/// Tests that all Boolean values are correctly loaded at
/// their default settings upon start of simulation
/// </summary>
[TestMethod]
public void testBooleanRuleValues()
{
    Assert.AreEqual(true, FlockingBirds.CohesionOn);
    Assert.AreEqual(true, FlockingBirds.SeparationOn);
    Assert.AreEqual(true, FlockingBirds.AlignmentOn);
    Assert.AreEqual(true, FlockingBirds.ObstacleOn);
    Assert.AreEqual(true, FlockingBirds.PredatorOn);
}


/// <summary>
/// Tests euclidean distance between two birds with positive
/// and negative co-ordinates
/// </summary>
[TestMethod]
public void EuclideanDistanceBirdClass()
{
    int result = (int)bird3.EuclideanDistance(bird2);
    Assert.AreEqual(377, result);
```

```
        int res1 = (int)bird4.EuclideanDistance(bird12);
        Assert.AreEqual(156, res1);
        int res2 = (int)bird11.EuclideanDistance(bird10);
        Assert.AreEqual(0, res2);
        int res3 = (int)bird7.EuclideanDistance(bird9);
        Assert.AreEqual(85105, res3);
        int res4 = (int)bird4.EuclideanDistance(bird5);
        Assert.AreEqual(325, res4);
    }

    /// <summary>
    /// Tests Euclidean distance between bird and static obstacle
    /// </summary>
    [TestMethod]
    public void EuclideanDistanceObstacleClass()
    {
        int result = (int)bird7.EuclideanDistanceObs(obstacle2);
        Assert.AreEqual(86, result);
        int res = (int)bird6.EuclideanDistanceObs(obstacle1);
        Assert.AreEqual(41850, res);        }


    /// <summary>
    /// Tests Euclidean distance between bird and set of vector
    /// co-ordinates
    /// </summary>
    [TestMethod]
    public void EuclideanDistanceVectorTest()
    {
        Vector2 myVec = new Vector2(548, 129);
        int result = (int)bird8.EuclideanDistanceVectors(myVec);
        Assert.AreEqual(397, result);
    }

    /// <summary>
    /// Tests Euclidean distance between bird and set of vector
    /// co-ordinates
    /// </summary>
    [TestMethod]
    public void EuclideanDistanceVectorTest2()
    {
        Vector2 myVec = new Vector2(-654, -45745);
        int result = (int)bird9.EuclideanDistanceVectors(myVec);
        Assert.AreEqual(131155, result);
    }

    /// <summary>
    /// Tests the Bird class constructor
    /// </summary>
    [TestMethod]
    public void BirdConstructor()
    {
        Assert.AreEqual(bird10.BirdPosition.X, 300);
        Assert.AreEqual(bird10.BirdPosition.Y, 400);
    }

    /// <summary>
    /// Tests the Obstacle class constructor
    /// </summary>
    [TestMethod]
    public void ObstacleConstructor()
    {
        Assert.AreEqual(obstacle.ObstaclePosition.X, 642);
        Assert.AreEqual(obstacle.ObstaclePosition.Y, 385);
```

```
}

/// <summary>
/// Tests the Bird class constructor using negative values for
/// bird position
/// </summary>
[TestMethod]
public void BirdConstructorNegValues()
{
    Assert.AreEqual(bird1.BirdPosition.X, -250);
    Assert.AreEqual(bird1.BirdPosition.Y, -400);
}

/// <summary>
/// Tests the BirdDistanceController function
/// </summary>
[TestMethod]
public void BirdDistanceControllerTest()
{
    bird11.BirdDistanceController(FlockingBirds);
    Assert.AreEqual(3200, bird11.BirdDestination.X);
    Assert.AreEqual(1700, bird11.BirdDestination.Y);

    bird12.BirdDistanceController(FlockingBirds);
    Assert.AreEqual(-2200, bird12.BirdDestination.X);
    Assert.AreEqual(-2399, bird12.BirdDestination.Y);
}

/// <summary>
/// Tests the RandomNumberGenerator class and function
/// </summary>
[TestMethod]
public void RandomNumberGeneratorCheck()
{
    Assert.IsTrue(4 > RandomNumberGenerator.Next(4));
    Assert.IsTrue(-1 < RandomNumberGenerator.Next(50));

    bool result = false;
    int num = (int)RandomNumberGenerator.Next(2);
    if (num == 0 || num == 1)
        result = true;
    Assert.IsTrue(result);
}

/// <summary>
/// Tests vector maths to check correct mathematical assumptions
/// for the XNA framework
/// </summary>
[TestMethod]
public void VectorChecker()
{
    Vector2 vec1 = new Vector2(200, 150);
    Vector2 vec2 = new Vector2(30, 50);
    Vector2 vec10 = new Vector2(0, 0);
    Vector2 vecA = new Vector2(100, 200);
    Vector2 vecB = new Vector2(120, 200);
    Vector2 vec3, vec4, vec5, vec6, vec7, vec9;
    Vector2 vec13, vec14, vec15, vec16, vecAFin, vecBFin;

    vec3 = vec1 + vec2;
    vec4 = vec1 - vec2;
    vec5 = vec2 - vec1;
    vec6 = vec1 * vec2;
    vec7 = vec1 / vecA;
```

```
        vec9 = vec3 / 2;

        Assert.AreEqual(230, vec3.X);
        Assert.AreEqual(200, vec3.Y);
        Assert.AreEqual(170, vec4.X);
        Assert.AreEqual(100, vec4.Y);
        Assert.AreEqual(-170, vec5.X);
        Assert.AreEqual(-100, vec5.Y);
        Assert.AreEqual(6000, vec6.X);
        Assert.AreEqual(7500, vec6.Y);
        Assert.AreEqual(2, vec7.X);
        Assert.AreEqual(0.75, vec7.Y);
        Assert.AreEqual(115, vec9.X);
        Assert.AreEqual(100, vec9.Y);

        //These check the vector maths on the Separation Rule, which
        //is replicated here.
        vec13 = (vec10 - (vecB - vecA));
        vec15 = vecA + vec13;
        vecAFin = vec15;
        Assert.AreEqual(80, vecAFin.X);
        Assert.AreEqual(200, vecAFin.Y);

        vec14 = (vec10 - (vecAFin - vecB));
        vec16 = vecB + vec14;
        vecBFin = vec16;
        Assert.AreEqual(160, vecBFin.X);
        Assert.AreEqual(200, vecBFin.Y);
    }

    /// <summary>
    /// Tests the AngleBirdFacing function
    /// </summary>
    [TestMethod]
    public void AngleBirdFacingTest()
    {
        //Radians returned
        double oneAngRad = one.AngleBirdFacing();
        double twoAngRad = two.AngleBirdFacing();
        double threeAngRad = three.AngleBirdFacing();
        double fourAngRad = four.AngleBirdFacing();

        Assert.AreEqual(2.8301413636869981, oneAngRad);
        Assert.AreEqual(1.1120991756080376, twoAngRad);
        Assert.AreEqual(0.070886946568588183, threeAngRad);
        Assert.AreEqual(2.4106535494308297, fourAngRad);
    }

    /// <summary>
    /// Tests AngleBetweenBirds function
    /// </summary>
    [TestMethod]
    public void AngleBetweenBirdsTest()
    {
        int res1 = (int)one.AngleBetweenBirds(bird13);
        Assert.AreEqual(69, res1);
        int res2 = (int)bird13.AngleBetweenBirds(bird7);
        Assert.AreEqual(74, res2);
        int res3 = (int)bird7.AngleBetweenBirds(bird14);
        Assert.AreEqual(-12, res3);
        int res4 = (int)bird14.AngleBetweenBirds(one);
        Assert.AreEqual(11, res4);
    }
```

```
        /// <summary>
        /// Tests AngleBetweenBirdAndObstacle function
        /// </summary>
        [TestMethod]
        public void AngleBetweenBirdAndObstacleTest()
        {
            int res1 = (int)one.AngleBetweenBirdObstacle(obstacle);
            int res2 = (int)three.AngleBetweenBirdObstacle(obstacle1);
            int res3 = (int)bird12.AngleBetweenBirdObstacle(obstacle2);
            Assert.AreEqual(53, res1);
            Assert.AreEqual(3, res2);
            Assert.AreEqual(106, res3);
        }

        /// <summary>
        /// Tests conversion of radians into degrees for mathematical
        /// calculations
        /// </summary>
        [TestMethod]
        public void RadiansToDegreeTest()
        {
            //Radians returned
            double oneAngRad = one.AngleBirdFacing();
            double twoAngRad = two.AngleBirdFacing();
            double threeAngRad = three.AngleBirdFacing();
            double fourAngRad = four.AngleBirdFacing();

            //Converts radians to degrees
            int oneAngDeg = (int)(oneAngRad * 180 / Math.PI);
            int twoAngDeg = (int)(twoAngRad * 180 / Math.PI);
            int threeAngDeg = (int)(threeAngRad * 180 / Math.PI);
            int fourAngDeg = (int)(fourAngRad * 180 / Math.PI);

            Assert.AreEqual(162, oneAngDeg);
            Assert.AreEqual(63, twoAngDeg);
            Assert.AreEqual(4, threeAngDeg);
            Assert.AreEqual(138, fourAngDeg);
        }
    }
}
```

APPENDIX F

Code Generated by XNA Game Studio when Creating a New Windows Game:

The following code is generated automatically within Visual Studio 2008 using XNA Framework 3.0 when a new “Windows Game” project is created. This is a basic skeleton upon which the programmer builds, adding commands as required.

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace WindowsGame1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }

    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to
        /// before starting to run.
        /// This is where it can query for any required services and
        /// load any non-graphic related content.
        /// Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
```

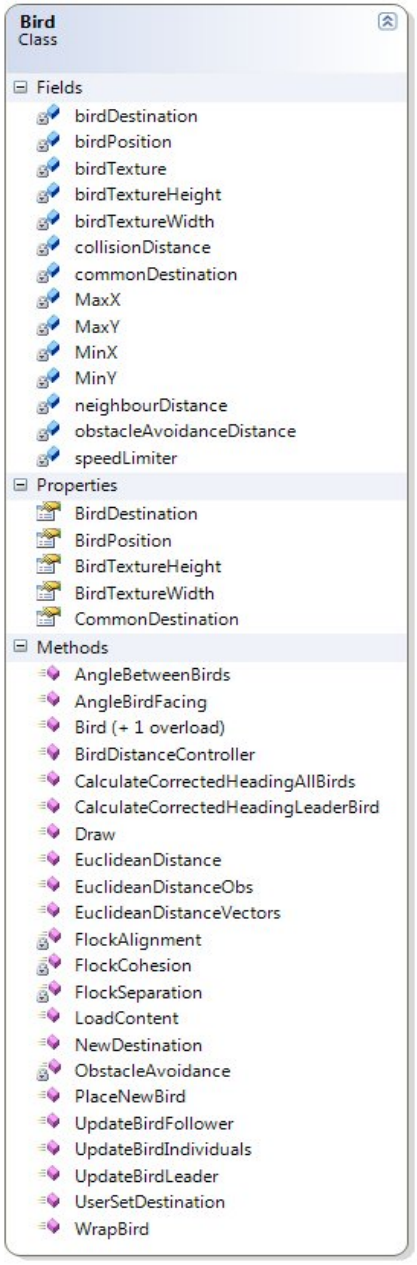
```
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the
        /// place to load all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw
               textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the
        /// place to unload all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.
        /// </param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
                  ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.
        /// </param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
    }
}
```

APPENDIX G

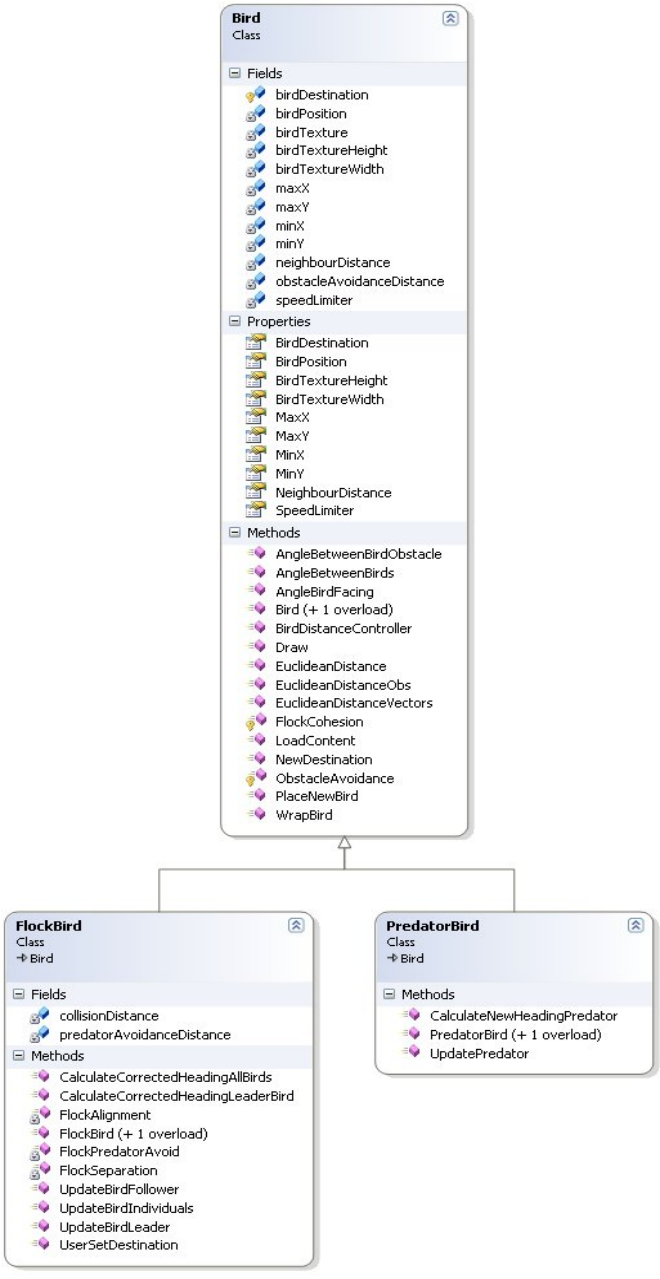
Class Diagrams for Refactored Bird Class:

With the introduction of a Predator Bird, the Bird class was refactored from a single class into a Bird base class with inherited classes of FlockBird and PredatorBird. Below are the class diagrams for this refactoring of the Bird class – first when it was a single class, and then once it had been split into base class with inherited classes. The majority of variables and functions are common to both types of birds so they remain in the base class to be used by all inherited classes, with only elements specific to each sub-class being moved down into the inherited classes.

Single Bird Class:



Bird Base Class with Inherited Flock Bird and Predator Bird:



APPENDIX H

Notes For The CD:

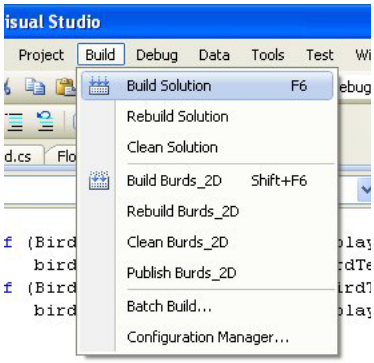
H.1 Running the Simulation:

Unfortunately due to restrictions with the current full release version of the XNA framework it is not as yet possible to create a standalone executable application for this simulation. In order to run it, it is necessary to have installed upon your machine both Microsoft Visual Studio 2008 as well as the XNA Game Studio 3.0 CTP (Community Technology Preview). Visual Studio 2008 is available on some of the lab computers and as a free download within the college from the Elms site, and the XNA Framework is available as a free download from Microsoft and has been included on this CD for your convenience. Visual Studio has not been included as this is not generally free software and its inclusion therefore may be considered as piracy.

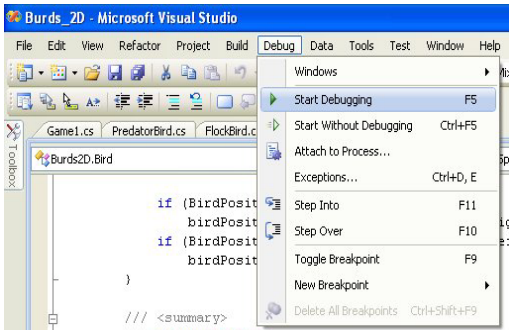
If it is not possible or desirable to install these onto your machine the Systems Group have offered to assist in any way they can, or alternatively contact Dr Keith Mannock who I believe has the required software installed or available and should be able to run the programme for you.

Alternatively please feel free to contact me and I would be delighted to attend the college to demonstrate the simulation for you on a suitable machine.

If running the simulation in Visual Studio 2008, please open the **Burds_2D.sln** file (in the Burds_2D directory).



Go to the Build menu and select “Build Solution”.



Once it is built go to Debug and select “Start Debugging”. This will run the simulation.

Once it is running, follow GUI instructions for use of the simulation. To end, click on the familiar red cross in the top right corner.

H.2 What's on the CD:

- **Code for the "Burds" Simulation:** The solution including test suite for the project may be opened as outlined above in Visual Studio 2008 with XNA 3.0 CTP. This is in the folder named "Code for Simulation".

- **XNA Game Studio 3.0 CTP (Community Technology Preview):** This is a free download from Microsoft so has been included onto the CD for easy installation. Visual Studio 2008 is required to install the XNA framework.

- **Project Report in both PDF and Microsoft Word format**

- **Screen Captures (Movies) in AVI format of simulation running:** In the event that it is not possible to view the actual simulation due to lack of appropriate software, several screen capture movies have been prepared for viewing of different aspects of the simulation running.

  PLEASE NOTE: The movies show a somewhat jerky simulation which is a direct result of the screen capture software running concurrently with the simulation and the computer's inability to run smoothly under these conditions. When running normally the simulation is not jerky but delivers very smooth graceful and flowing movement of the birds. The movies are therefore no substitution for the real simulation but are included to provide a general idea of the birds behaviour under various conditions and rule combinations.

  Full list of movies available is provided in appendix section H.3 (following page).

- **Software to play AVI movies:** Microsoft Media Player 11 is included which should enable playback of AVI movies. If this does not allow playback, the codec belonging to the screen capture software has also been included (CamStudioCodec10.zip) as well as the actual screen capture software itself (CamStudio20.exe). Both of these are open source and freely available for personal use and have been included on this CD with full documentation from their original source.

- **Project Proposal Form:** This is the original proposal form for the project.

H.3 List of Movies

The movies all have the following options selected unless otherwise stated:
Birds have individual destinations (not following a leader bird)
Obstacle Avoidance ON
Random Destination Change OFF
Speed set at default value
Predator is ON when it is visible on the screen. Some movies have predator present for approximately half the time and off for the rest so that behaviour may be observed both with and without it. This will be obvious to the viewer.
Where Predator is ON, it "eats" birds and they are removed from the simulation.
In many movies more birds are added to the simulation by user when numbers get low (for example when eaten by predator).
Status of all rules can be viewed in bottom left corner of screen.

1. FullSimulation1 / FullSimulation2 / FullSimulation3:

These show the simulation with all steering rules ON (cohesion, separation and alignment) and the user adding more birds when numbers get low (having been eaten by the predator).

2. CohesionOnly:

Cohesion rule ON, Separation and Alignment OFF.
Birds swirl around in groups but do not travel effectively.

3. SeparationOnly:

Separation ON, Cohesion and Alignment OFF.
Birds move around screen in random destinations avoiding collisions but not moving together at all.

4. AlignmentOnly:

Alignment ON, Cohesion and Separation OFF.
Birds make no attempt to separate from each other or to join together into groups or flocks. The alignment rule makes them travel in the same general direction.

5. Cohesion&SeparationOnly:

Cohesion and Separation ON, Alignment OFF.
Birds move around in groups, staying together and not colliding, but with no effective direction or heading.

6. Cohesion&AlignmentOnly:

Cohesion and Alignment ON, Separation OFF.

Birds try to get as close to each other as possible with no separation rule to keep them apart. Move with a direction or heading.

7. Separation&AlignmentOnly:

Separation and Alignment ON, Cohesion OFF.

Birds move in the same general direction as each other, aligning themselves with their neighbours, and avoid collisions but make no attempt to flock together.

8. PredatorOff&RandomDestChange:

All steering rules ON. Predator bird OFF. Random Destination Change for all birds ON.

Birds will collectively change direction at randomly timed intervals. This looks a little like the flock "sneezing" as they appear to pause briefly and all point in different directions before deciding on a collective heading and moving off again.

9. UserPlacedTargets:

All steering rules ON. User places target destinations on screen.

The pointing finger is the user placed target, which causes all birds to initially change their destinations towards those co-ordinates.

10. ObstaclePlacement:

All steering rules ON. Starts with only one bird and predator. More birds added by user.

Obstacles placed on screen by user.

11. ObstacleAvoidanceOff:

All steering rules ON. Obstacle Avoidance OFF.

Birds move around screen completely ignoring obstacles.

12. SpeedVariations:

All steering rules ON.

Speed of simulation altered by user. First decreased to slowest speed, then incrementally increased to full speed before returning to "normal" default speed.

13. FollowingLeaderBird:

Birds follow a leader bird and do not have individual destinations.

All other rules ON. Behaviour is poor with this option, with birds becoming so crowded that they collide often.

14. PauseSimulation:

All steering rules ON.

Simulation paused and recommenced several times to demonstrate the pause facility.

APPENDIX I

Online tutorials accessed to learn XNA and C# Game Programming:

Fegelein.com (2006). Microsoft XNA Framework; Loading a 3d model. [Online] Available at http://www.fegelein.com/?p=15 Accessed 9th July 2008.

Fegelein.com (2006). Microsoft XNA Framework; Drawing a Complex Mesh. [Online] Available at http://www.fegelein.com Accessed 9th July 2008.

Nuclex. (2007). Using XNA to draw a rotating triangle. [Online] Available at http://www.nuclex.org/articles/using-xna-to-draw-a-rotating-triangle Accessed 9th July 2008.

MSDN XNA Developers Centre (2008). [Online] Available at http://msdn.microsoft.com/en-us/library/bb200104.aspx Accessed from 30th May 2008 onwards.

MSDN XNA Developers Centre (2008). Your First Game: Microsoft XNA Game Studio in 2D. [Online] Available at http://msdn.microsoft.com/en-us/library/bb203893.aspx Accessed from 30th May 2008 onwards.

MSDN XNA Developers Centre (2008). Going Beyond: XNA Game Studio in 3D. [Online] Available at http://msdn.microsoft.com/en-us/library/bb203897.aspx Accessed from 30th May onwards.

Riemers XNA Tutorials (2008). XNA and DirectX Tutorials. [Online] Available at http://www.riemers.net/eng/tutorials.php Accessed from 1st June onwards.