Simulating Bird Flocking in 2D

Object-Oriented Programming using C++

Max Elliott - 1434717

17/01/2017
School of Physics and Astronomy
University of Birmingham
Birmingham, B15 2TT

1. Introduction

In 1986, Craig Reynolds created the first computer simulation to model complex animal flocking behaviours[1]. The simulation used large groups of independent agents known as boids who followed simple behavioural rules which, when left to follow these rules, convincingly generated flocking patterns seen in real groups of animals.
The aim of this project was to use the same flocking rules proposed by Reynolds to simulate 2D bird flocking. The code was written using C++ in the Qt Creator IDE. A particular focus was put on using object-oriented programming techniques when writing the code. Extra features have been implemented, such as predators and obstacles, to emphasise these programming concepts.

2. Methodology

The idea of using independent agents following individual rules to model more complex systems has become a staple method for computer modelling. In this case, each agent will be a Bird in a collection of Birds known as a Flock. Each bird in the flock will follow simple behavioural algorithms to generate the complex flocking behaviour. For each frame of the simulation when running, each bird in the flock will update its movement based on these rules, and then move based on this update.

2.1 Behavioural Rules

The three rules initially implemented by Reynolds will serve as a basis for each bird's behaviour. The three rules are as follows:

1) Cohesion - each bird will steer itself towards the average position of neighbouring birds
2) Alignment - each bird will try to align itself with the average direction of neighbouring birds i.e birds will try to match their velocity to that of neighbouring birds
3) Separation - birds will avoid getting too close to each other

In addition to these rules, three further behavioural rules will be applied for each Bird:

1) Fear - an extra class of Bird known as a Predator will be introduced. Predators will chase the nearest bird. Birds will flee away from any neighbouring predators.
2) Wall avoidance - the simulation will be shown visually in a square graphical display. Each wall of the display will therefore have a repulsive force to deter birds from flying off the screen.
3) Obstacle avoidance - one further feature will be the addition of static, circular obstacles that the birds must avoid while flocking. This avoidance will be simulated with a force that steers birds to the sides of obstacles as they fly towards them.

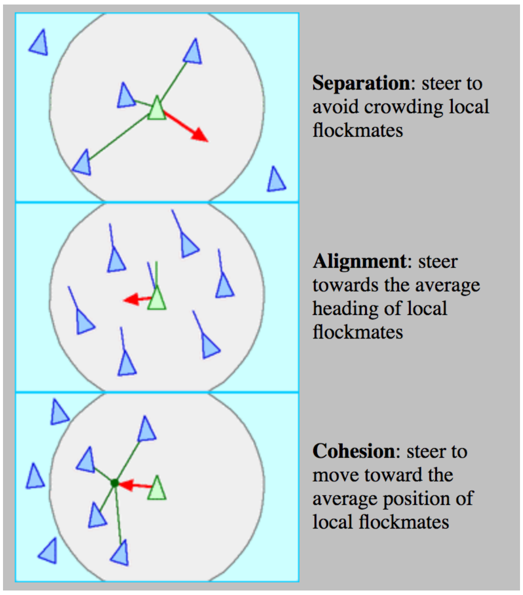

Separation: steer to avoid crowding local flockmates
Alignment: steer towards the average heading of local flockmates
Cohesion: steer to move toward the average position of local flockmates

figure 1. visual representation of the three basic behaviours [1].

To implement these rules, each bird must obviously have a position and velocity, which in this case will be a 2-dimensional vector. Each bird will also have a detection radius and separation radius. The detection radius will define how far the bird can 'see' around them; any bird in this radius will be counted as a neighbour. The separation distance will be used for the separation rule, and birds will repel each other at this distance.

2.2 Object-Oriented Methodologies

Below are some key concepts of object-oriented programmed that will be used in this project, with an aim of improving the code design.

2.2.1 Classes

Object-oriented programming is implemented through classes. Classes can be seen as blueprints that define the data members and methods for objects used in a program. A class generally has a header file which is the interface of the class. It declares the data members and methods of the class. The implementation of the methods however is generally written in a .cpp file of the same name as the header.
Instantiations of classes can be created, called objects, by calling a class's constructor. When a constructor is called, memory is assigned for that object's data, either on the main heap or on the calling method's local stack depending on how it is called. An object will have its own instances of the data members, and can call any methods defined in its class. Object-oriented programming is centred around using these classes to create objects that can then interact with each other to achieve the results the code is aiming for.

2.2.2 Inheritance and Hierarchy

Inheritance and hierarchy are powerful tools in object-oriented programming for improving code design and efficiency. A class can be defined as a subclass of another class (termed the superclass). When this is done, the subclass will inherit the data members of its parent, as well as all of its public and protected methods. Therefore any instantiation of a subclass object can call any of these methods itself.

By implementing inheritance to create a hierarchy of classes, you can exploit code reuse; if a group of classes all require a certain set of data members or methods, then creating a baseclass and setting them all to inherit from it will reduce duplicate code that would otherwise have to be written individually for each class. This improves code design, and if part of a method needs to be changed or another data member needs to be added, it can simply be done in the bass class rather than all the classes.

3. Code Implementation

This section focuses on how the methodologies were practically applied in the program code. The program can be categorised into three sections: the flock objects, the flock, and the user interface. Each of these sections will be discussed, with particular attention being paid to the usage of object-oriented techniques.

3.1 Flock Objects

These are a group of classes that each correspond to the different objects in a flock that will interact with each other. The classes are Obstacle, Bird and Predator. Each bird/predator/obstacle in the simulation will be an instantiation of its corresponding class. A simple flowchart is shown below to show how each classes inherits from each other.
Each class is, either directly or indirectly, a subclass of the FlockObject base class. The FlockObject class contains two vital data members all subclasses need to have for the program to work.

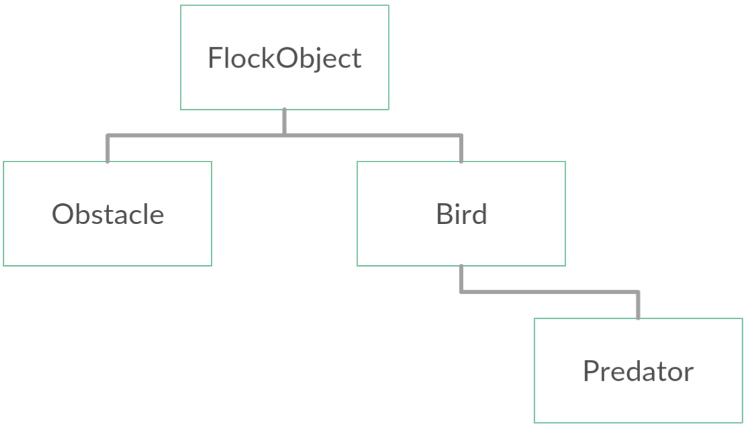
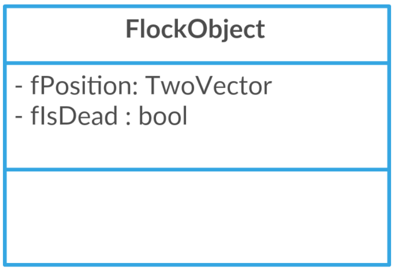


figure 2. Hierarchy of class inheritance from FlockObject

fPosition is required for all flock objects to interact properly as all flocking rules are dependent on relative positions of objects. It is a TwoVector, a class inspired by Antonio Sergi's ThreeVector class, which represents a 2-dimensional vector. The class defines many methods and operators used to manipulate vectors, which prove useful when implementing the behavioural rules. The origin of the coordinate system is the top-left corner of the display window, as this is how it is defined when displaying graphics.

fIsDead is a boolean that indicates whether an object is 'dead' or not, and is set to true for objects that need to be removed from the simulation.



FlockObject

- fPosition: TwoVector
- fIsDead : bool

figure 3. UML class diagram of FlockObject.

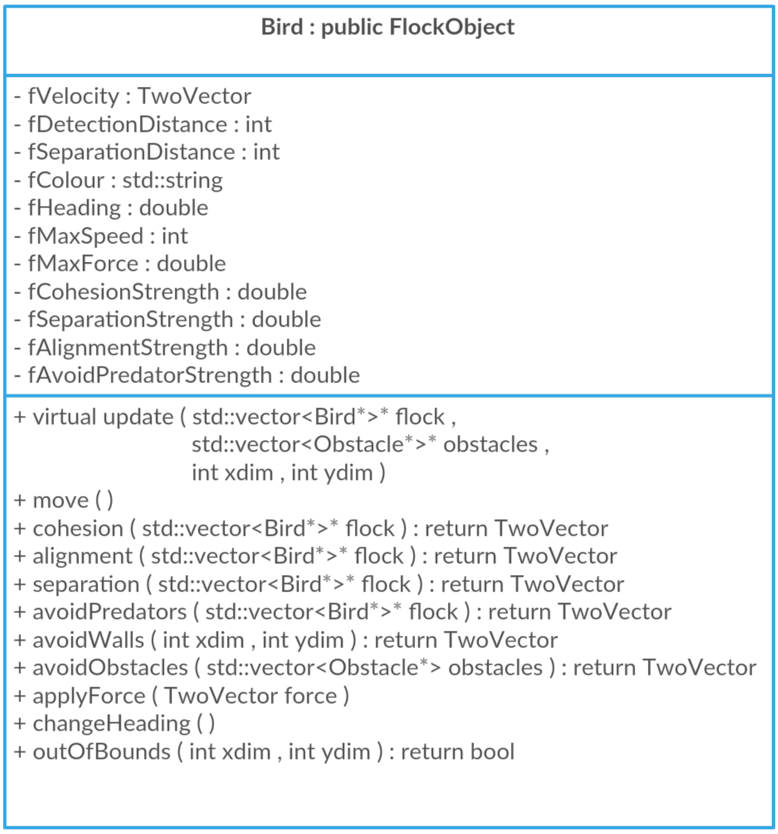
Note All classes inherit publicly, meaning they can access all public methods and data members defined in their parent classes. However, all data members are declared private in each class, as is good general coding practice. This means subclasses cannot directly access these inherited data members. Therefore all data members will have getter and setter methods to retrieve and alter them, so that inherited classes can still manipulate inherited data members. While all classes have getters and setters for their data members, they have been neglected from the UML class diagrams in this report to improve readability.

3.1.1 Obstacle

The simplest subclass of FlockObject is the Obstacle class. It inherits publicly from FlockObject. This class is used to represent obstacles the Birds have to avoid. Obstacles are circular and have a radius defined by fRadius.

3.1.2 Bird

This is the class used to define the birds in the flock. Each bird is an instantiation of this class. They inherit the data members of FlockObject and also have many newly declared data members.



Bird : public FlockObject

- fVelocity : TwoVector
- fDetectionDistance : int
- fSeparationDistance : int
- fColour : std::string
- fHeading : double
- fMaxSpeed : int
- fMaxForce : double
- fCohesionStrength : double
- fSeparationStrength : double
- fAlignmentStrength : double
- fAvoidPredatorStrength : double

```
+ virtual update ( std::vector<Bird*>* flock ,
                   std::vector<Obstacle*>* obstacles ,
                   int xdim , int ydim )
+ move ( )
+ cohesion ( std::vector<Bird*>* flock ) : return TwoVector
+ alignment ( std::vector<Bird*>* flock ) : return TwoVector
+ separation ( std::vector<Bird*>* flock ) : return TwoVector
+ avoidPredators ( std::vector<Bird*>* flock ) : return TwoVector
+ avoidWalls ( int xdim , int ydim ) : return TwoVector
+ avoidObstacles ( std::vector<Obstacle*> obstacles ) : return TwoVector
+ applyForce ( TwoVector force )
+ changeHeading ( )
+ outOfBounds ( int xdim , int ydim ) : return bool
```

- the bird's velocity
- detection radius
- separation radius
- colour of the bird
- direction of velocity (in degrees from x-axis)
- maximum allowed speed
- maximum magnitude of a behavioural force
- relative weightings of behavioural forces

- updates the bird's velocity based on its behaviour
- moves the bird (adds velocity to position)
- force towards average position of neighbours
- force to align with average velocity of neighbours
- force to move away from bird in separation radius
- force to flee from predators
- force to move away from walls
- force to move away from obstacles
- applies the forces to the bird's velocity
- calculates the bird's heading from its velocity
- checks if the bird has managed to leave the display screen

figure 4. UML class diagram of Bird.

The major method for the simulation in Bird is *update*, which updates the Bird's velocity using its behavioural rules. *update* follows this pseudocode:

```
void update(*flock, *obstacles, int xdim, int ydim){
   sep = separation(flock);                    //calculate TwoVector for each
   coh = cohesion(flock);                      //behavioural method.
   ali = alignment(flock);
   wall = avoidWalls(xdim, ydim);
   pred = avoidPredators(flock);
   obs = avoidObstacles(obstacles);

   sep = sep * sepWeighting;                   //weight vectors by relative
   coh = coh * cohWeighting;                   //strengths, so certain forces
   ali = ali * aliWeighting;                   /have more influence
   wall = wall * wallWeighting;
   pred = pred * predWeighting;
   obs = obs * obsWeighting;

   if (outOfBounds(xdim, ydim) setIsDead(true);//remove bird if off the display

   applyForce(sep+coh+ali+wall+pred+obs);      //update bird's velocity
}
```

figure 5. *Bird::update* pseudocode.

It takes as arguments a collection of all birds and obstacles in the flock, as well as the dimensions of the display window, as they are needed for the behaviours of the bird. Each behaviour is implemented as an individual method which is then called in *update*. Each behavioural method returns a TwoVector for how the bird should change its fVelocity due to that behaviour. It can be seen as a force being applied to the Bird [2]. Each behaviour calculates its force via this pseudocode:

```
TwoVector behaviouralRule(…){
  desiredVector = GenerateDesiredVector();     //how this vector is generated
                                                 depends on the particular rule

  desiredVector.weight(genericWeighting);      //apply general weighting

  steeringVector = desiredVector - velocityVector; //calculates change in
                                                   //velocity needed to reach
                                                   //desiredVector

  if(steeringVector.magnitude > fMaxForce){    //limit force magnitude to
      limit to maxForce;                       //improve realism
  }


  return steeringVector;                       //returns force to apply later
}
```

figure 6. *Bird::behaviouralRule* pseudocode.

A desiredVector is found e.g. a vector from the bird to the average position of its neighbours for cohesion, or a vector representing the average velocity of neighbours for alignment. This vector represents the desired velocity of the Bird due to that behaviour. However, returning just the desiredVector would lead to unrealistic movement in the form of quick, instantaneous changes in speed and direction. Instead of using this vector directly, a 'steering' force is calculated, which corresponds to the vector the bird's velocity should change by to move towards its desired position.

Each steering vector is then weighted in *update*, so that certain forces will affect the Bird's movement more than others. For instance, it makes sense that birds will be more concerned about fleeing from predators than aligning with neighbours.
The vectors returned by each behavioural method are then simply summed together to create an acceleration which is applied via the *applyForce* method, whose pseudocode is as follows:

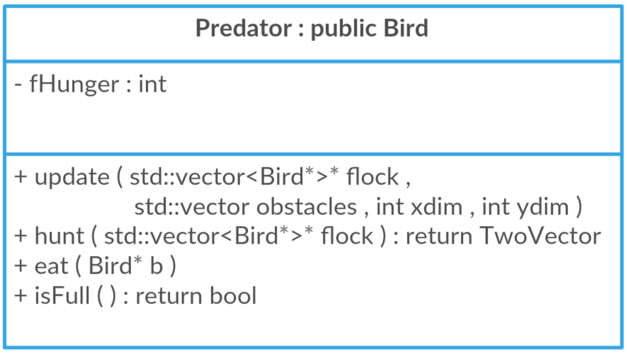
```
void applyForce(TwoVector force){
   if(force.magnitude == 0){                //if no change due to behavioural
      velocity = velocity*1.01;             //rules, just accelerate slightly
   }
   else{                                    //add behavioural forces
      velocity = velocity + force;
   }

   if(velocity.magnitude > maxSpeed){       //limit speed to fMaxSpeed
      limit to maxSpeed;
   }

   changeHeading();                         //calculate new heading from velocity
}
```

figure 7. *Bird::applyForce* pseudocode.

To actually move a bird, the *move* method needs to be called after *update*, that simply adds fVelocity to fPosition.

3.1.3 Predator

The Predator class defines the predators that will chase the birds. Predator is a subclass of Bird as both share a lot of similar data members and both will be simulated in the same way: an *update* method to apply the behavioural rules, and a *move* method to move them. Therefore, Predator can simply inherit Bird's data members and methods instead of duplicating code.
Predator reimplements the *update* method, which follows the same pseudocode as in the Bird class, but it applies different behavioural methods instead. It applies all behaviours (inherited from Bird) except *cohesion* or *alignment*, as it made sense that predators didn't want to group together. It also applies a newly defined behavioural method *hunt*, which cycles through all birds, and finds the closest one in the predator's detection radius and chases it. If a predator is close enough to the bird, it will eat it and set the bird's fIsDead to true. Predators also have a new data member, fHunger, that sets how many birds they will eat before disappearing.



Predator : public Bird

- fHunger : int

+ update (std::vector<Bird*>* flock ,
  std::vector obstacles , int xdim , int ydim)
+ hunt (std::vector<Bird*>* flock) : return TwoVector
+ eat (Bird* b)
+ isFull ( ) : return bool

- amount of birds the predator can eat

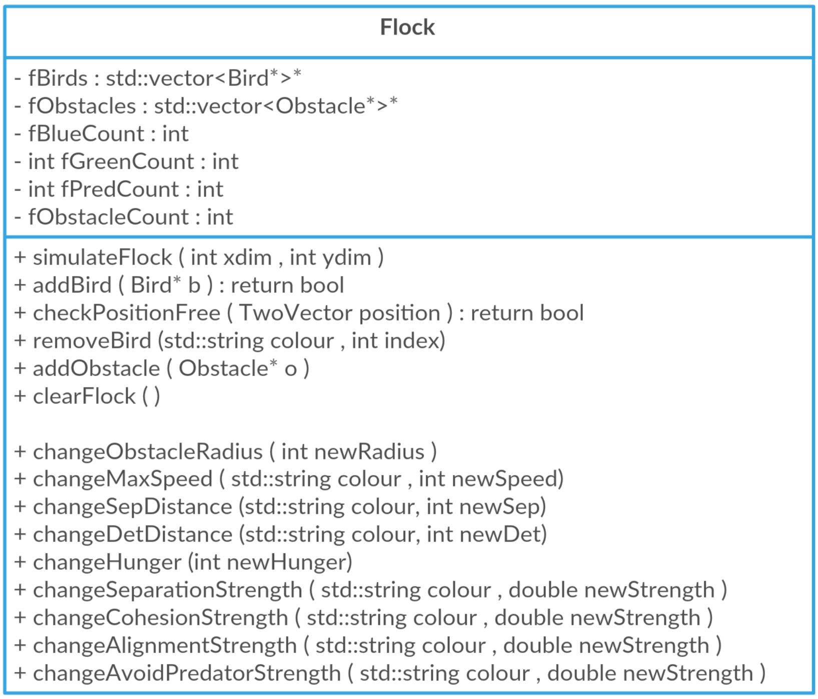
- update Predator's velocity based on its behaviour

- behavioural force to chase the nearest bird
- removes the bird b if it is caught
- checks if fHunger equals 0, and sets the Predator's fIsDead to true is it is

figure 8. UML class diagram of Predator. Note that data members and methods from Bird do not have to be redeclared, as Predator publicly inherits from Bird.

3.2 Flock

Flock is a class that manages all objects in the simulation and controls the main bulk of the simulation process.



```
Flock

- fBirds : std::vector<Bird*>*
- fObstacles : std::vector<Obstacle*>*
- fBlueCount : int
- int fGreenCount : int
- int fPredCount : int
- fObstacleCount : int

+ simulateFlock ( int xdim , int ydim )
+ addBird ( Bird* b ) : return bool
+ checkPositionFree ( TwoVector position ) : return bool
+ removeBird (std::string colour , int index)
+ addObstacle ( Obstacle* o )
+ clearFlock ( )

+ changeObstacleRadius ( int newRadius )
+ changeMaxSpeed ( std::string colour , int newSpeed)
+ changeSepDistance (std::string colour, int newSep)
+ changeDetDistance (std::string colour, int newDet)
+ changeHunger (int newHunger)
+ changeSeparationStrength ( std::string colour , double newStrength )
+ changeCohesionStrength ( std::string colour , double newStrength )
+ changeAlignmentStrength ( std::string colour , double newStrength )
+ changeAvoidPredatorStrength ( std::string colour , double newStrength )
```

- all instances of Bird in the simulation
- all instances of Obstacle in the simulation
- number of blue birds
- number of green birds
- number of predators

- updates all birds/predators, and removes any dead birds
- adds a bird/predator to the flock
- checks if a bird is being added inside an obstacle
- removes specific element from fBirds
- adds an obstacle to the flock
- clears fBirds and fObstacles

- methods to changes parameters of FlockObjects. Colour is used to specify which birds to modify.

figure 9. UML class diagram of Flock.

fBirds is a std::vector that contains all instances of Bird currently in the simulation, including all Predators (because Predator inherits from Bird, a Predator can be used anywhere where a Bird is required). fObstacles contains all instances of Obstacle currently in the simulation. The std::vectors are defined as pointers, and each element is also a pointer. This means all objects are stored in memory on the heap, rather than the local stack. This is done for two reasons:

1) By declaring the elements as pointers, each element can be stored anywhere in memory and be retrieved with just the memory address. The alternative would be to store all elements in order, which could require a large block of free memory.
2) By declaring the vector as a pointer, only the pointer needs to be passed as an argument in methods that need the vector, such as the *update* method in Bird. If it weren't a pointer, the whole vector would have to be copied into each Bird's local stack for each *update*, which would greatly affect run-time performance.

For the simulation to progress, each instance of Bird in fBirds needs to have its *update* and *move* methods called. This is done in the *simulateFlock* method of the Flock class. The pseudocode is as follows:

```
simulateFlock( int xdim, int ydim ){

   for ( int i=0; i < fBirds.size(); i++ ){      //updates all birds
      Bird* b = fBirds.at(i);
      b.update( fBirds, fObstacles, xdim, ydim );//passes fBirds, fObstacles
                                                 //and display dimensions for
                                                 //the behavioural methods.

      if( b.isDead() ){                          //removes dead birds
         removeBird(b);
      }
   }

   for each Bird in fBirds{                      //move birds
      b.move();
   }

   for (int i=0; i < fObstacles.size(); i++){    //remove dead obstacles
      Obstacle* o = fObstacle.at(i);
      if( o.isDead() ){
         removeObstacle(o);
      }
   }
}
```

figure 10. *Flock::simulateFlock* pseudocode

It cycles through each Bird, and runs their *update* and *move* methods, passing the pointers to fBirds and fObstacles to each Bird, as well as the dimensions of the display window, as the behavioural methods require them. The *simulate* method also removes all dead objects from the simulation.

This is where the effects of polymorphism are shown. fBirds is a std::vector of type Bird but, as Predator is a subclass of Bird, it can contain both Birds and Predators. When a Predator is stored in fBirds, its pointer type will still be Bird. However, due to *update* being declared *virtual* in Bird, when it is called for each element, the program dynamically choses which *update* method to call at run-time. For a Bird the *update* method in the Bird class is called, but for a Predator it calls the *update* method as defined in the Predator class.

Flock contains methods to add/remove Bird/Predator/Obstacle objects to the simulation. It also contains methods to change the parameters of the Birds in fBirds. The parameters can be changed separately for each bird colour.

3.3 User Interface

The main part of the program, the part responsible for the simulation, is essentially self-contained in the Flock class and the FlockObject classes. All that is needed to run the simulation is something to create an instantiation of Flock, populate it with flock objects using the methods in Flock, and then repeatedly call the *simulateFlock* method. A graphical user interface (GUI) was created in Qt Creator to achieve this. The user interface is split into two separate windows: the control window and the display window. Therefore, two classes were needed: MainWindow and DisplayWindow.

3.3.1 MainWindow

This class is the ‘central hub’ of the program. It contains an instantiation of the Flock class to encompass the simulation aspects, and provides methods to add and remove FlockObjects from the simulation. It also has an instantiation of DisplayWindow to provide a graphical view of the simulation. MainWindow uses a QTimer to call its *simulate* function every 20ms. This calls the *simulateFlock* method in fFlock, to run the simulation, and passes x_dimension and y_dimension as arguments.

As well as the slot for running the simulation, MainWindow defines a private function, known as a slot, for when each slider in the interface is moved, when the count boxes are changed, and when the pause/reset buttons are pressed. When a signal is produced by a slider, it will run the connected slot's code.

3.3.2 DisplayWindow

The DisplayWindow class is used to create the window that displays the flock as it is simulated. It is a subclass of QWidget, a Qt Creator class. The constructor of DisplayWindow takes a pointer to the Flock object that is being simulated as an argument. A QTimer is then used to call *update* every 20ms. *update* is a function DisplayWindow inherits from QWidget, and is an optimised way of calling the *paintEvent* method. *paintEvent* creates a QPainter object, defined by the Qt Creator library, to draw all FlockObjects on screen. It does this by cycling through fBirds and fObstacles in its fFlock data member, generating a shape to be drawn for each FlockObject based on its fPosition, and then drawing it in the correct colour on screen.

3.4 Run-time Procedure

The overall process the program uses to simulate the bird flock is as follows:

1) *main.cpp* runs its main method. This calls the MainWindow constructor to create an instance of it, and shows it on screen.
2) The MainWindow constructor creates a Flock and DisplayWindow object by calling their constructors, and fills the fBirds data member in the Flock with 100 instances of the Bird class. The DisplayWindow is shown on screen.
3) The MainWindow uses a QTimer to run its *simulate* function every 20ms. *simulate* runs the simulation by calling *Flock::simulateFlock*, which in turn calls *Bird::update* on every Bird in fBirds, or *Predator::update* if it is a Predator. *simulateFlock* then moves the Birds by calling *Bird::move* on each. *simulateFlock* also checks and removes dead birds or obstacles.
4) At the same time, *QWidget::update* in the DisplayWindow is called every 20ms to draw the simulation onscreen. *QWidget::update* takes the flock and paints every Bird in fBirds and every Obstacle in fObstacles, using *DisplayWindow::paintEvent*.
5) Birds, Predators and Obstacle objects can be added or removed from fBirds and fObstacles in the flock through the MainWindow GUI, and the settings of each object can be changed. This is done by using slot functions, that are connected to the signals sent from the GUI controls. These slots call the appropriate methods in Flock.

Steps 3 and 4 are represented visually using UML activity diagrams in figures 12 and 11 respectively.

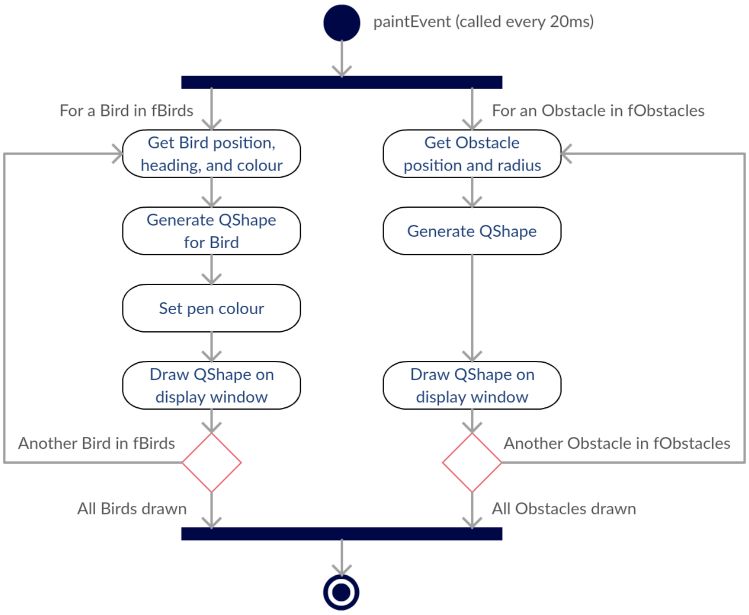

paintEvent (called every 20ms)
For a Bird in fBirds
For an Obstacle in fObstacles
Get Bird position, heading, and colour
Get Obstacle position and radius
Generate QShape for Bird
Generate QShape
Set pen colour
Draw QShape on display window
Draw QShape on display window
Another Bird in fBirds
Another Obstacle in fObstacles
All Birds drawn
All Obstacles drawn

figure 11. Activity diagram of the paintEvent method, that is called every 20ms in the DisplayWindow.

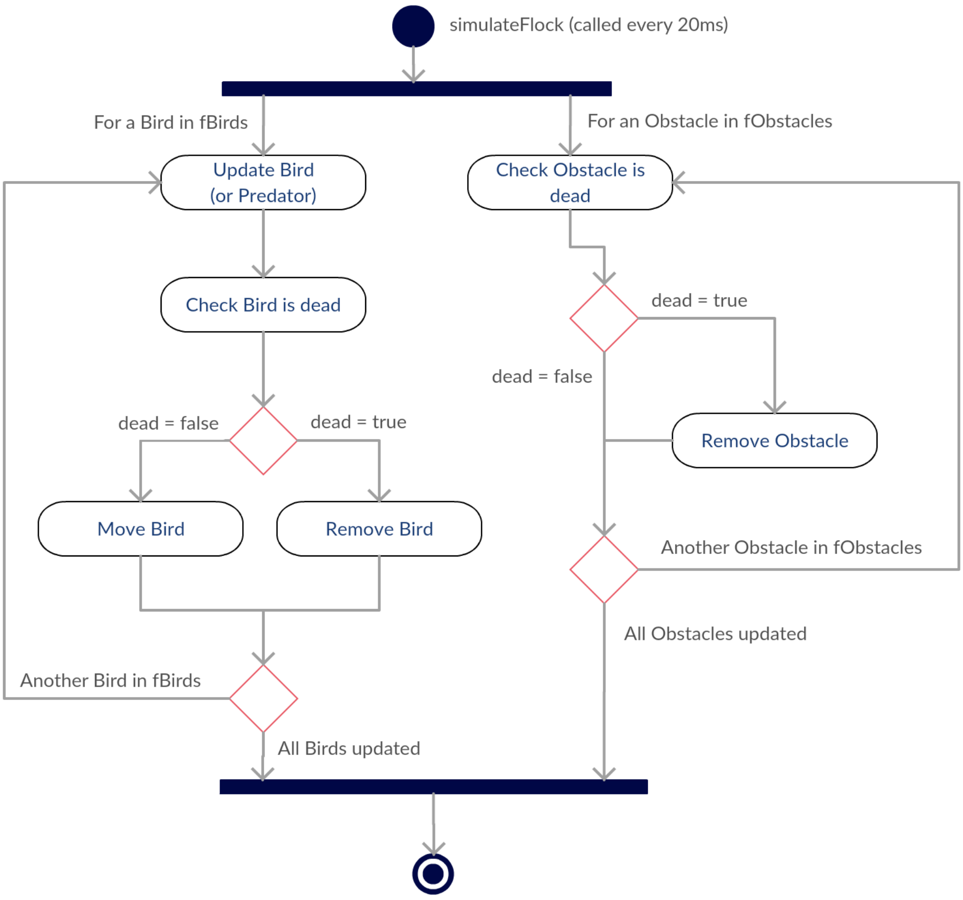


figure 12. Activity diagram of the *simulateFlock* method, that is called every 20ms in the MainWindow method *MainWindow::simulate*.

4. Overview of Finished Program

This section shows the finished program as the user sees it, and gives an overview of all the features that were implemented. The user interface is split into two windows, the main window and the display window.

The display window is where the flock is graphically shown. Birds are shown as triangles and obstacles as black rings. Birds come in two different colours: blue and green. A bird will only choose to flock with its same colour, and will try to separate form the other colour. Red triangles are the predators, and will chase birds of other colours to try and eat them. Once eaten, a bird will be removed from the simulation. Predators have a hunger i.e. the amount of birds they will eat. Once this limit is reached, they too are removed. All birds and predators will stay on the display window, and the window can be resized as desired.

The main window contains the controls for the simulation. Each bird colour has its own controls, consisting of a count box to add and remove birds, and sliders to change their max speed, separation radius, and detection radius. Changing these sliders will apply the change to all birds of that colour. Predators have an extra slider for their hunger.
The window also has an ‘advanced settings’ tab, where the relative strength of each basic behavioural rule can be changed. The strengths can be different for each colour bird. The initial settings are designed to show realistic, but different, flocking behaviours for each colour.
Obstacles have a count box to add and remove them, and a radius slider to set the size of them. Birds will avoid obstacles, but if they are to accidentally hit one, the bird will be removed.
When adding a new bird, it will be added at a random position on the display, facing in a random direction. Birds cannot be added inside obstacles.
There is a pause button to pause the simulation, and a reset button. The reset button removes all current birds and obstacles, and resets the flock to 50 green birds and 50 blue birds. All settings are returned to the default values.

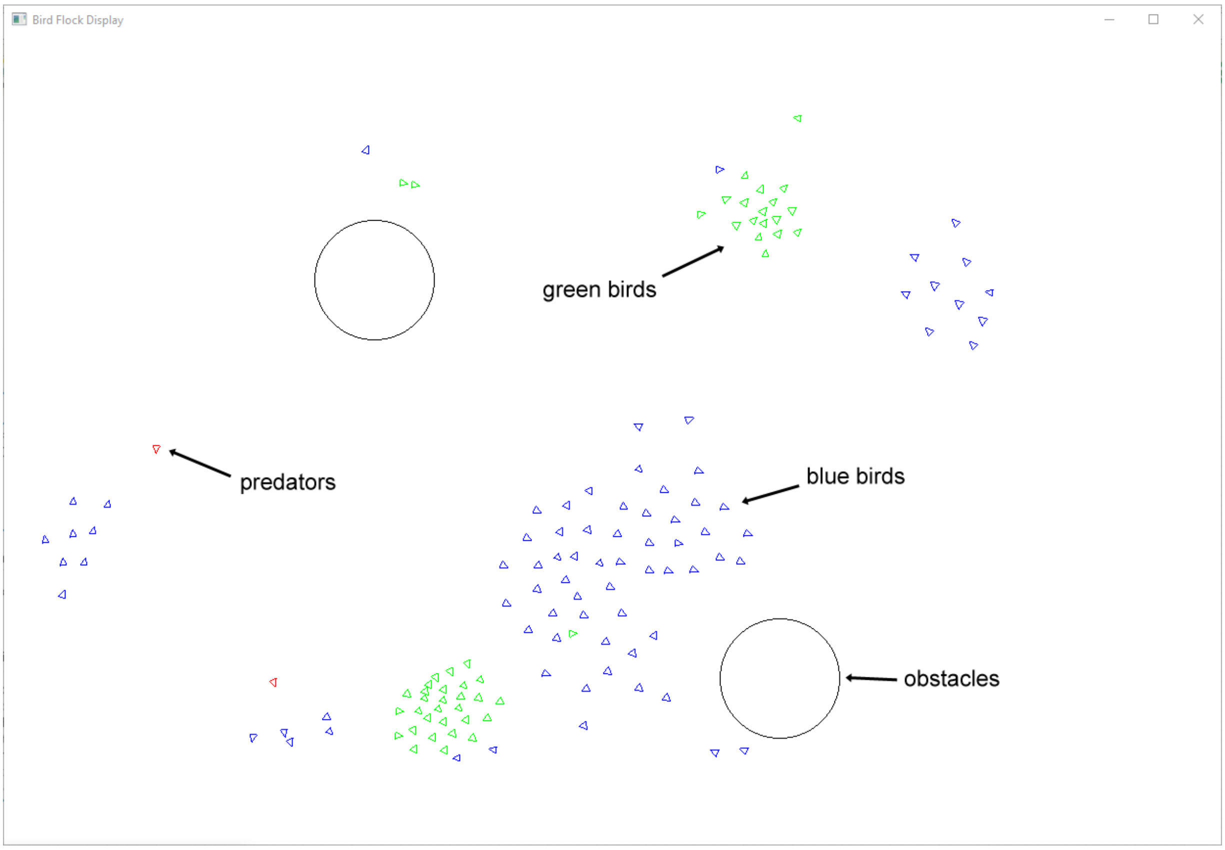
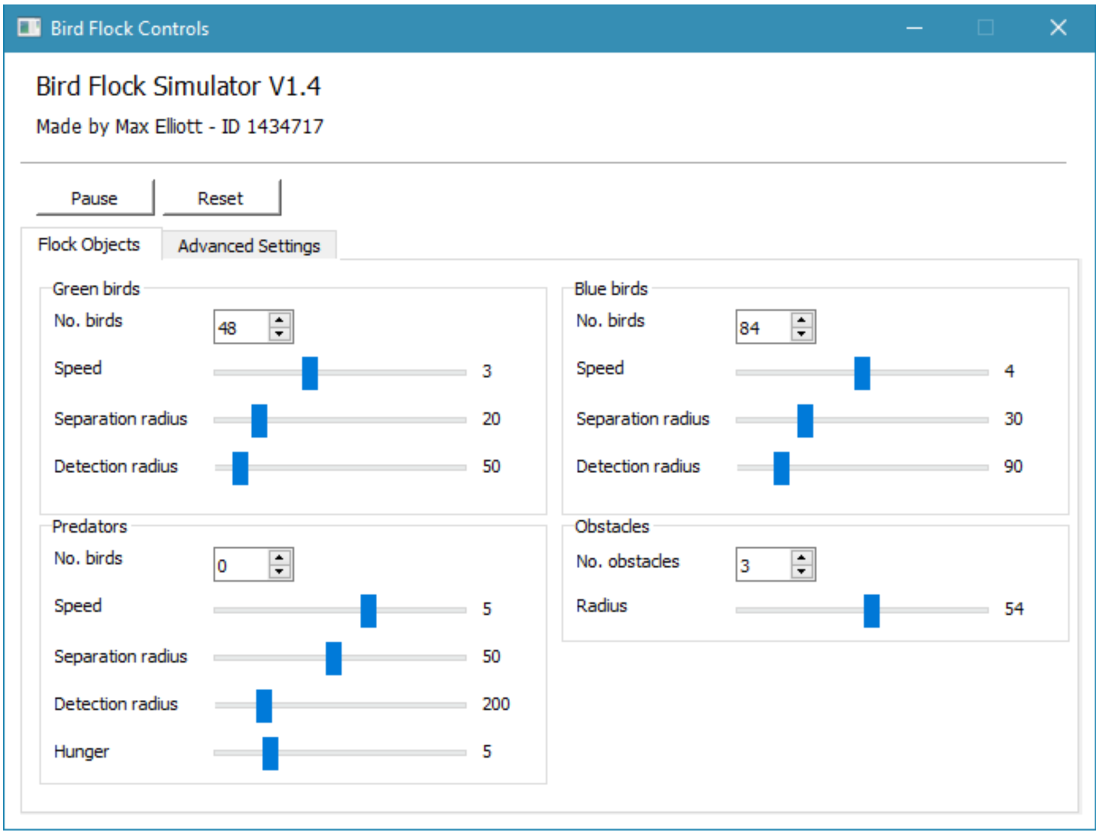


figure 13. The display window when the simulation is running.


Bird Flock Controls

Bird Flock Simulator V1.4

Made by Max Elliott - ID 1434717

Green birds

No. birds 48

Speed 3

Separation radius 20

Detection radius 50

Blue birds

No. birds 84

Speed 4

Separation radius 30

Detection radius 90

Predators

No. birds 0

Speed 5

Separation radius 50

Detection radius 200

Hunger 5

Obstacles

No. obstacles 3

Radius 54

figure 14. The main window of the program that contains all controls.

5. Analysis and Improvements

This program seems to model bird flocking convincingly, with the birds interacting with each other in a realistic way. The birds will group together, flee from predators who begin chasing them, and avoid obstacles especially well. Turning the maximum speed of birds up too high will cause them to occasionally hit the obstacles or escape the display window. Certain configurations of overlapping obstacles will also cause the birds to hit them, which is a problem but not a significant one. The GUI gives the user a large amount of control. By altering the detection/separation distances and the relative strengths of the behavioural forces, a variety of group behaviours can be shown e.g. small groups of closely-packed birds, or much larger and broader flocks.

The program can simulate a flock of approximately 250 birds without obvious performance issues, but beyond that the program starts to slow. While the code has generally been designed to be efficient at run-time, improvements could still be made. For instance, each of the three basic behavioural methods in Bird cycle through the entire flock vector, resulting in three cycles. This could be reduced to one single method that applied all three rules at the same time, requiring only one cycle through the vector. However, even with these changes, the time complexity of the simulation would still be $O(n^2)$, where n is the number of birds in the simulation, as each bird has to cycle through all other birds to apply the behavioural rules. To make a significant difference to the performance, this time complexity class needs to be reduced.

A possible way to reduce the time complexity would be split the simulation into a grid. The Flock class would contain a std::vector of birds for each grid box, that contains all the Birds in that box. Assuming the grid spacing was comparable to the birds' detection radius, a bird would only have to cycle through the birds in its box plus the adjacent boxes when applying the behavioural rules. When the birds are evenly distributed amongst the grid, this would prove to be a significant performance enhancer. However, as the birds started to group into similar grid boxes, the performance gains would be reduced.

6. Conclusion

A program to simulate bird flocking in 2D was created using C++, based on the flocking behavioural rules proposed by Craig Reynolds. Additionally, predators were introduced that the birds fled from, and obstacles can be added that the birds must avoid. The bird parameters can be adjusted through a user interface, created using Qt Creator libraries.

The program was created using object-oriented methodologies, which have been thoroughly discussed. These methodologies allowed the program to be clearly segmented and resulted in efficient and readable code. The Flock class holds collections of Bird/Predator/Obstacle objects that are then simulated using behavioural methods defined in the object classes themselves. UI classes are used as an interface for the user to control different aspects of the flock. Because of this object-oriented design, it would be easy to add further subclasses of the Bird class to represent differently behaving birds (analogous to the Predator class), or subclasses of FlockObject for other objects (e.g. bird food the birds try to eat).

The simulation proved to be successful, producing convincing flock behaviour. The program performs well up to 250 birds, and methods to improve run-time performance have been discussed.

References

1) http://www.red3d.com/cwr/boids/

2) *Steering Behaviours for Autonomous Characters*, Craig Reynolds , 1999