Birds Flocking in Three Dimensions
Using Object-Oriented Programming Techniques to Simulate the Interactions of Birds

Isobel Romero-Shaw

Date: December 9, 2016
Student ID: 1353719

Abstract

The motion of birds can be categorised into the three key behaviours of flocking, fleeing, and avoidance. This behaviour is modelled in the *Birds* program described in this report. *Birds* was written in C++ using the Qt Creator integrated development environment, which enabled the creation of the graphical user interface of the program. The simulation is three-dimensional, with the z-component of an object's position indicated by the opacity of its visual representation. This report emphasises the use of object-oriented programming techniques within the program. The behavioural algorithms driving bird movements are compared to those in existing designs, with key differences highlighted. Future extensions to the program are postulated, with realistic suggestions of their implementation.

Contents

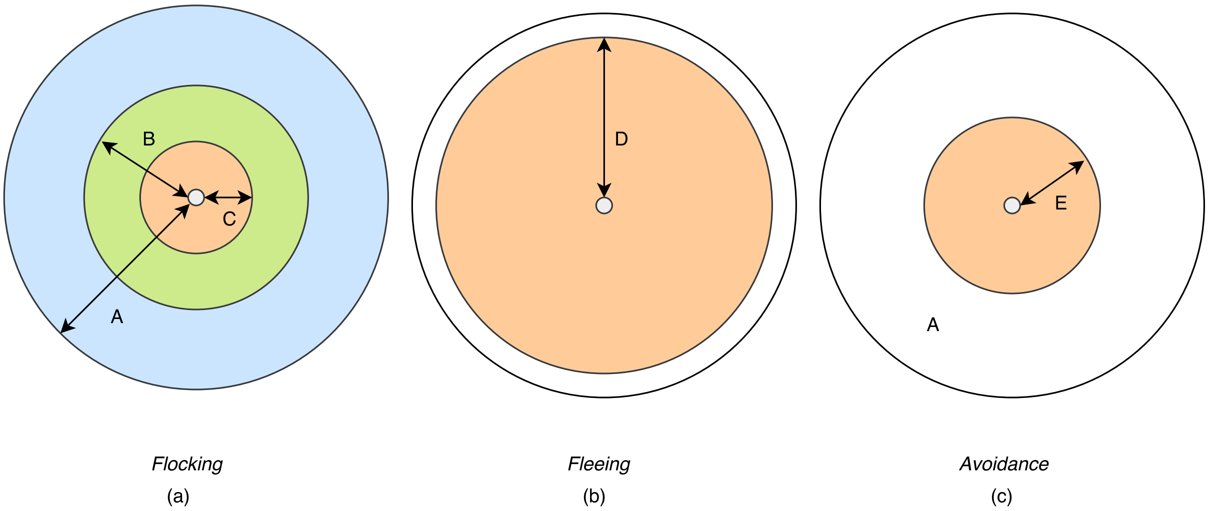

Figure 1: *Schematic representation of different ranges checked by prey bird in its three behavioural modes.*

1 Introduction

Flocking, swarming and schooling are often beautiful natural phenomena, with patterns forming as the result of many independent bodies reacting to stimuli within their individual locale. Flocking was first artificially simulated by Craig Reynolds in 1987 with the now-famous *Boids* program[1]. Since its conception, the computational logic behind these rules has produced many effective models of group behaviour.

In this report, an alternative flocking simulation is assessed. The mathematical formulae driving the bird motions are explained, and their translation into code demonstrated. Object-oriented programming techniques are described and exemplified by their usage within the program.

Birds was written in C++, using the Qt creator integrated development environment. A complete summary of all new classes, their public methods and private data members is provided in the appendix.

2 Theory

2.1 Prey Movement

Flocking arises from group behaviour without central co-ordination. Each bird acts within its individual sphere of awareness, represented in figure (1) as a circle centred on the bird position.
Recalling that

$$\mathbf{p} = m\Delta\mathbf{v} = m\frac{d\mathbf{x}}{dt}, \tag{1}$$

where m is the mass of the bird and $\Delta\mathbf{v}$ its change in velocity, it can be shown that for a bird of unit mass,

$$\frac{\mathbf{p}}{dt} = d\mathbf{x}. \tag{2}$$

Over a unit time interval,

$$\mathbf{p} = d\mathbf{x}, \qquad dt = 1. \tag{3}$$

Each bird adjusts its momentum based on its proximity to others. The incremented momentum and position are related by

$$\mathbf{p_2} = \mathbf{p_1} + d\mathbf{p}, \tag{4}$$
$$\mathbf{x_2} = \mathbf{x_1} + \mathbf{p_2}. \tag{5}$$

The way a bird moves depends on the objects within its sight range, A. It will flock with other prey, flee from predators and avoid obstacles. Each behaviour is weighted appropriately to the overall momentum change, which is divided by the number of objects within the system to reduce the accumulated reaction magnitude. These factors are combined within the $ScaleFactor$ term in the following code examples.

2.1.1 Flocking

Using the $FlockWith$ funciton, prey bird i will optimise its distance from another prey bird j. Position difference is categorised by the three distances shown in figure (1(a)).

1. $A \geq |\Delta\mathbf{x_{ij}}| \geq B$

   In this case, i moves towards j. Its momentum change, $\Delta\mathbf{p_i}$, depends on $\Delta\mathbf{x_{ij}}$, as illustrated by
   $$\Delta\mathbf{p} = |\frac{A}{|\mathbf{x_i}| - |\mathbf{x_j}|}|\Delta\mathbf{x_{ij}} \times 10^{-4}. \tag{6}$$

2. $B \geq |\Delta\mathbf{x_{ij}}| \geq C$

   In this case, $\mathbf{p_i} = (0.0, 0.0, 0.0)$. Distance B is equal to $\frac{A}{30}$.

3. $|\Delta\mathbf{x_{ij}}| \leq C$

   In this case, bird i steers away from j, with momentum change
   $$\Delta\mathbf{p} = -|\frac{C}{|\mathbf{x_i}| - |\mathbf{x_j}|}|\Delta\mathbf{x_{ij}} \times 10^{-4}. \tag{7}$$

   A preceding if statement in the implementation of these equations of motion prevents division by zero.

```
if (fabs(PosDiff) <= TooClose)
{
    if (PosDiff == 0.0)
{
    ScaleFactor *= 1e4;
    delMom = -this->GetMaxMom()*PosDiffVect*fabs(TooClose/PosDiff)*ScaleFactor;
}
else
{
      delMom = -PosDiffVect*fabs(TooClose/PosDiff)*ScaleFactor;
}
```

   Here, $PosDiff$ contains the scalar value $|\mathbf{x_i}| - |\mathbf{x_j}|$, $PosDiffVect$ the vector position difference $\Delta\mathbf{x_{ij}}$, $delMom = \Delta\mathbf{p}$, and $ScaleFactor$ encompasses the contribution of the reaction to the total momentum change. The shortest radius, C, is given by $TooClose = \frac{A}{50}$.

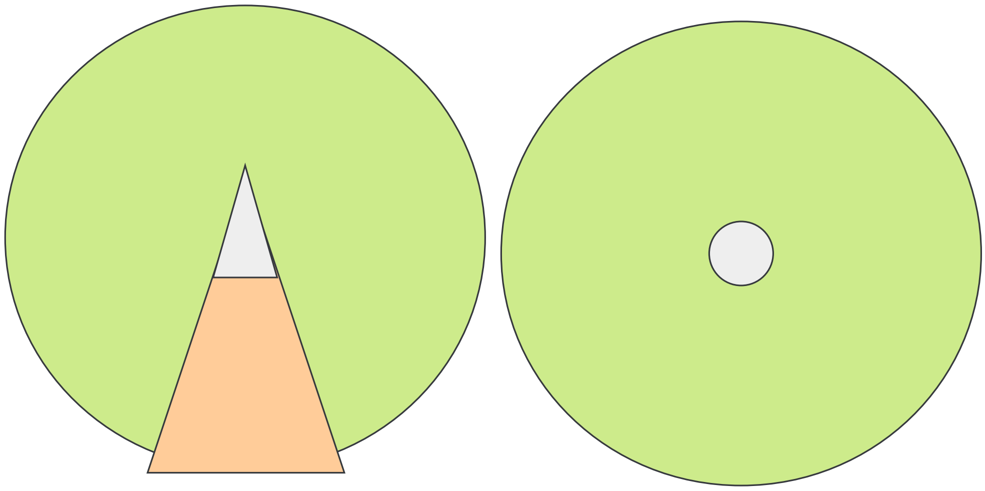

Figure 2: *Comparison of area of awareness for a Boid, left, and a Bird, right. Each agent interacts with other objects within the green area surrounding it.*

2.1.2 Fleeing

The momentum change due to a predator is larger, always dominating over the flocking movement.

$$\Delta\mathbf{p} = -|\frac{D}{|\mathbf{x_i}| - |\mathbf{x_j}|}|\Delta\mathbf{x_{ij}}\mathbf{p_{max}} \times 10^{-1}. \tag{8}$$

Distance $D = \frac{3A}{2}$.

2.1.3 Avoidance

The avoidance performed by birds is given by

$$\Delta\mathbf{p} = -|\frac{E}{|\mathbf{x_i}| - |\mathbf{x_j}|}|\Delta\mathbf{x_{ij}} \times 10^{-3}. \tag{9}$$

where $E = \frac{A}{6}$.

2.1.4 Goal Seeking

A desired outcome of any flocking simulation is that the flock perpetually follows some instinctive path. A flock goal was created as an invisible obstacle with constant speed. This was then passed to each prey bird as something with which to flock, causing the group to follow its path in the absence of other obstacles or predators.

2.2 Comparison of Alternative Flocking Logic

Reynolds' *Boids* differs from *Birds* predominantly in its use of angular direction. Each Boid is represented by an isosceles triangle, the most acute vertex of which is its 'head'. Reynolds defined a 'blind spot' directly behind each boid, leading to a greater probability of forwards motion. Each Boid rotates towards the average heading of its neighbours. In comparison, the birds in this program move under the influence of objects within their entire sphere of awareness, resulting in a less realistic model. The future introduction of'blind spots' is considered in section 6.

Birds also lacks the velocity matching of nearby prey birds seen in *Boids*. The initial implementation of velocity matching, performed for nearby prey with momentum $FriendMom$ with the line

```
this->SetMom((this->GetMom() + FriendMom)*(0.5));
```

was removed to increase the visually more appealing randomness within Prey motion.

2.3 Predator Movement

Predatory momentum change is given by

$$\Delta \mathbf{p} = \left|\frac{A}{|\mathbf{x_i}| - |\mathbf{x_j}|}\right| \Delta \mathbf{x_{ij}} \times 10^{-2}, \tag{10}$$

a calculation performed upon every call of the *Hunt* function.

2.4 Obstacle Movement

Obstacle motions can be user-modified during simulation using the control window sliders. Otherwise, obstacles move with constant speed. This can be expressed mathematically by setting $d\mathbf{p}$ constant in equation (4).

3 Program Explanation

3.1 Code Structure

The class structure of the program is displayed in figure (3). In addition to the classes pictured, the program uses the C++/C libraries **vector** and **cmath**. The latter is included by **ThreeVector**$_{[a]}$, a class provided by Dr. Antonino Sergi which was reduced to provide only relevant sections of code. **MainWindow** and **DisplayWindow** inherit from Qt libraries **QMainWindow** and **QWidget** respectively. Both use timing functions defined in **QTimer**. **QPainter** is also included by DisplayWindow in order to create the graphical display.

3.2 Object-Oriented Programming Techniques

Object-oriented programming (OOP) is based upon the four key concepts[3] described below.

3.2.1 Inheritance

In program structure diagrams throughout this report, classes connected to those above by solid arrows inherit methods and data members from those classes. BirdPlayer is the parent of the Bird and Obstacle classes. Bird has children Prey and Predator. Functions and variables shared between multiple child classes need implementing only once in their parent class. All public access and mutation methods defined in BirdPlayer are available for use by inheriting classes. The Prey constructor is shown below.

```
Prey::Prey(ThreeVector Mom,
           ThreeVector Pos,
           double MaxMom)
    :Bird(Friendly, Mom, Pos, MaxMom, Starling)
{
    SetSight(200.0);
}
```

Constructors are not inherited, but are called explicitly in the constructor of the inheriting class. The Prey constructor ensures that every prey object has a player type of *Friendly*, a bird type of *Starling*, and a sight range of 200.0 units. This is set using *SetSight*, a mutator method that Prey inherits from Bird.

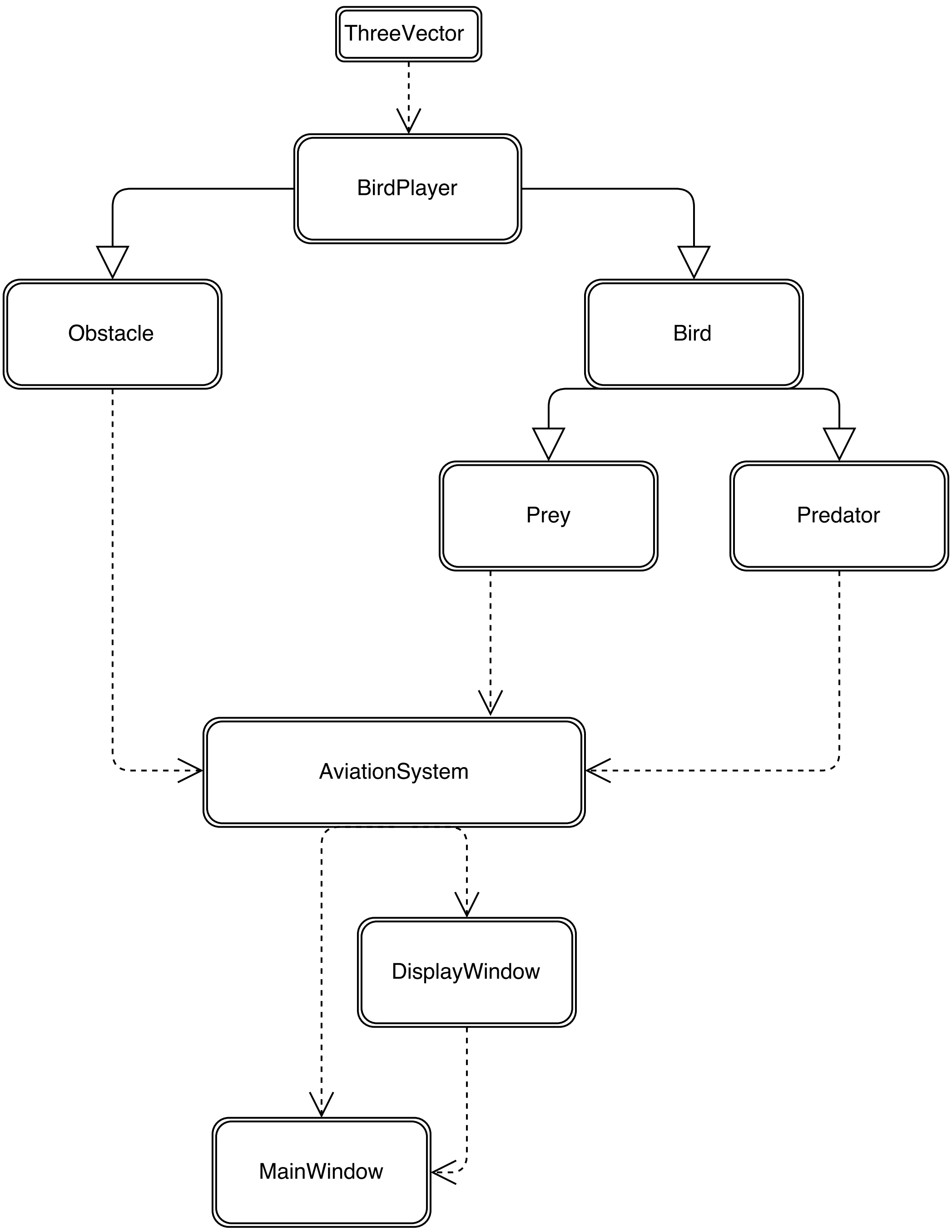

Figure 3: *Diagram of complete program structure. Inheritance relations are indicated by solid arrows, whilst inclusion relations are shown by dashed arrows.*

Virtual functions *GetPlayerType* and *Move* are declared in *BirdPlayer*. Setting them equal to 0 at declaration ensures that BirdPlayer is a pure virtual class, which is not instantiable. These methods are defined explicitly in the inheriting Obstacle class, making it a real class whose constructor may be used to create new instances of Obstacle. However, Bird is also a pure virtual class. It does not declare the above functions and has its own pure virtual method, *GetBirdType*. Real children Prey and Predator explicitly implement this in addition to *Move* and *GetPlayerType*. The shared functions correspond to different code sequences within each real class. This is known as polymorphism, the second key technique in OOP.

3.2.2 Polymorphism

Polymorphism allows the same commands to be applied to different objects that share the same base class. This is used in *Birds* to simplify procedures used to evolve the system and to display objects. In **AviationSystem**, three separate vectors contain pointers to Obstacle, Prey or Predator objects. A vector of pointers to BirdPlayer objects is also initialised. The first three vectors simplify user addition or removal of specific Obstacle, Prey or Predator objects, all of which evolve their position using *Move*. At simulation, the vector of BirdPlayer pointers is cleared and re-filled with those referenced by the other vectors, ensuring that it only ever includes pointers to existing objects. A simple code sequence can then be used to evolve the position of every player referenced by the pointers within the *fAllPlayers* vector.

```
if (fAllPlayers[i]->GetPlayerType() == BirdPlayer::Dangerous)
{
    fAllPlayers[i]->Move(fAllPlayers[i - NPrey], NAll);
```

NPrey is the size of the prey flock. All *Dangerous* players move with respect to the object referenced by pointer *fAllPlayers[i - NPrey]*. The predator then moves with respect to all non-prey objects in the system. Following this, the remaining vector elements are then iterated over, calling the *Move* method relevant to each referenced object. Similarly, within the *paintEvent* slot of the DisplayWindow class, a vector of BirdPlayer pointers, *fModelSystem*, is filled, iterated over, and cleared. Different pen styles are set for different player types, as demonstrated for a Prey bird below.

```
if (fModelSystem[i]->GetPlayerType() == BirdPlayer::Friendly)
{
    painter.setPen(preypen);
}
```

3.2.3 Encapsulation

Data members and methods relevant to a specific object are encapsulated in one defining class. Access restriction to members is controlled using the *public*, *protected* and *private* specifiers. Declarations within the *public* section of the header file are available to any other file within the program. *Protected* members can be accessed only by inheriting classes, and *private* members can be directly accessed only within the class of its declaration. The private data members of each class in *Birds* are shown in figure (4). Controlled access is permitted to private members only through use of public access and mutator methods.
Data encapsulation is linked to data abstraction, the final key aspect of OOP. Encapsulation stores all relevant data and procedures in a single class, whilst abstraction withholds all but the defining traits of an object.

3.2.4 Abstraction

The actions performed within any function are invisible at the level from which the function is called. In *Birds*, the *Simulate* command, which evolves the entire many-body system forwards in time, is called

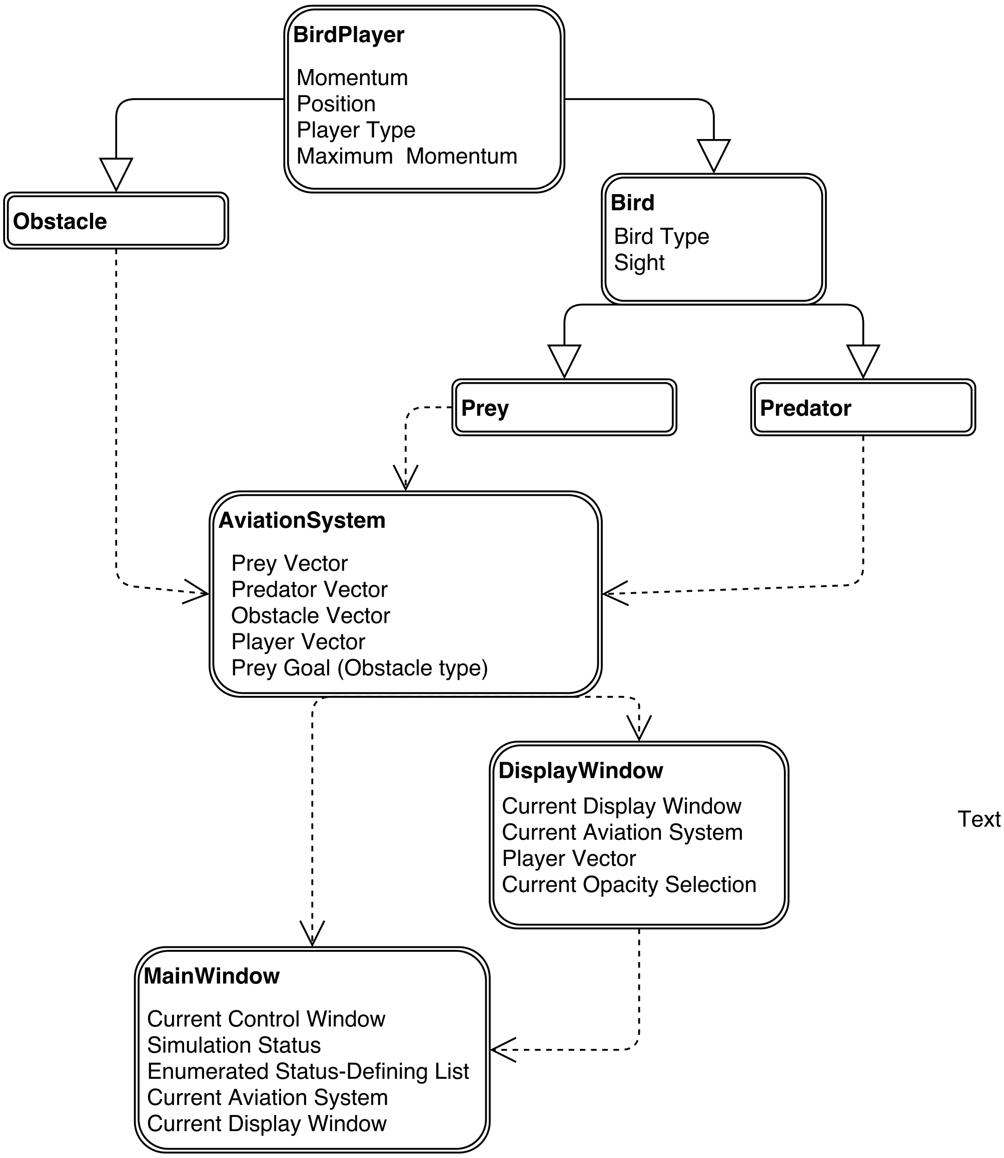

Figure 4: *Encapsulation of data within the **Birds** classes. The data implemented privately within each class is shown.*

compactly within MainWindow as shown.

```
void MainWindow::Animate()
{
    if(fRunStatus == Running)
    {
        fAviationSystem->Simulate();
    }
}
```

fRunStatus is an integer defining whether the simulation is active. When the *Run* button is clicked, this is changed from $Idle = 0$ to $Running = 1$.

3.3 Position and Momentum Restrictions

Object momentum is constrained to its maximum, and position limited to within the boundaries of the virtual space.
The maximum BirdPlayer momentum is regulated using the *RestrictMom* function. This finds the sign of each component of the current momentum vector, $\mathbf{p_c}$, and multiplies the result by the maximum player momentum, p_{max}. The restricted momentum, $\mathbf{p_r}$, is given by

$$\mathbf{p_r} = \frac{\mathbf{p_c}}{|p_c|} p_{max} \tag{11}$$

The implementation of the x-co-ordinate restriction will be demonstrated. The system dimensions are defined by macros in the header file of BirdPlayer.

```
#define Width 800.0
```

The compiler substitutes the numerical string associated with a macro wherever it finds the macro word in the program. The *BoundaryConditions* function facilitates elastic collisions with the boundaries of this virtual space. Demonstrating this, the procedure constraining the x-position is

```
double Origin = 0.0;
if (fPos.X() > Width)
{
    fPos.SetX(Width);
    fMom.SetX(-fabs(fMom.X()));
}
else if (fPos.X() < Origin)
{
    fPos.SetX(Origin);
    fMom.SetX(fabs(fMom.X()));
}
```

in which the x-component of the object's momentum switches sign at either end of the x-axis, and its position is constrained within 0.0 and $Width$. *BoundaryConditions* and *RestrictMom* are both called by the *UpdateAndRestrict* function, which is subsequently used within all movement methods.

4 Graphical User Interface

4.1 Functionality of User Interface

Birds features one graphical display window showing all objects. The user may influence the system using the control window, shown in figure (5).

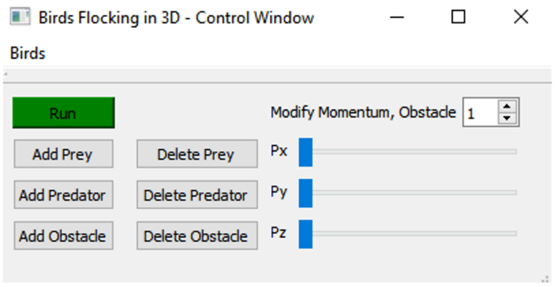

Figure 5: *Initial appearance of Birds control window.*

Figure 6: *Initial appearance of Birds display window.*

Upon clicking the *Run* button, the display window opens. The system initially has 1 moving obstacle and 15 prey, as shown in figure (6). In default simulation mode, the z position co-ordinate of an object is represented by its opacity, with maximum z making the object image opaque, and minimum z causing transparency. Un-checking the box labelled *Add Depth* in the lower right-hand corner of the display window projects all z-axis components onto the $x - y$ plane, full opacity of all images.
Each object has random initial x- and y-position co-ordinates and maximum z-value co-ordinate, ensuring that new objects are always visible. Birds have zero initial momentum, whilst obstacles are initialised with a random momentum vector.
The user may add or remove objects using control window buttons. Individual obstacles can be selected using the roll-box, and their momentum adjusted using sliders. Each slider corresponds to the particular momentum component indicated on the control window. Whilst simulating, the *Run* button becomes a red *Stop* button, terminating the display window when clicked.

4.2 Creation of User Interface

Qt Creator provides wizards that simplified the creation of the *Birds* user interface.

4.2.1 Signals and Slots

When interacted with, widget components send **signals** to their associated class. **Slots** are functions that executing upon receipt of a signal.

4.2.2 Main Window

The MainWindow constructor instantiates a new AviationSystem, sets the slider and spin-box ranges, window position, dimensions, title, and *Run* button appearance. Clicking *Run* sets simulation state to *Running*, changes the button colour to red, switches its text to *Stop*, and creates a new DisplayWindow. The second click resets the initial states and terminates the DisplayWindow. At construction, a timer is set to trigger *Animate* every 1×10^{-3} seconds.

4.2.3 Display Window

The DisplayWindow constructor sets its initial appearance, opacity state, window position and dimensions, which is set using the virtual space defined in BirdPlayer. A timer is also initialised within the constructor, causing the display screen to update every 1×10^{-3} seconds.

5 Comparison of Planned and Final Model

A variety of program structures were considered, all including ThreeVector in their base class and using Qt widgets for the user interface. The three main steps leading to the final design are shown in figure (7). The AviationSystem class used in the exiting program combines the functionality of the **Simulation** and **Flock** classes shown.

Two alternate predatory hunting styles were tested. The first set the predator to chase the flock centre of mass, whilst the other induced the single-prey chase seen in the final model. The first was proven ineffective, as flock dispersion caused the predator to hover around a central point.

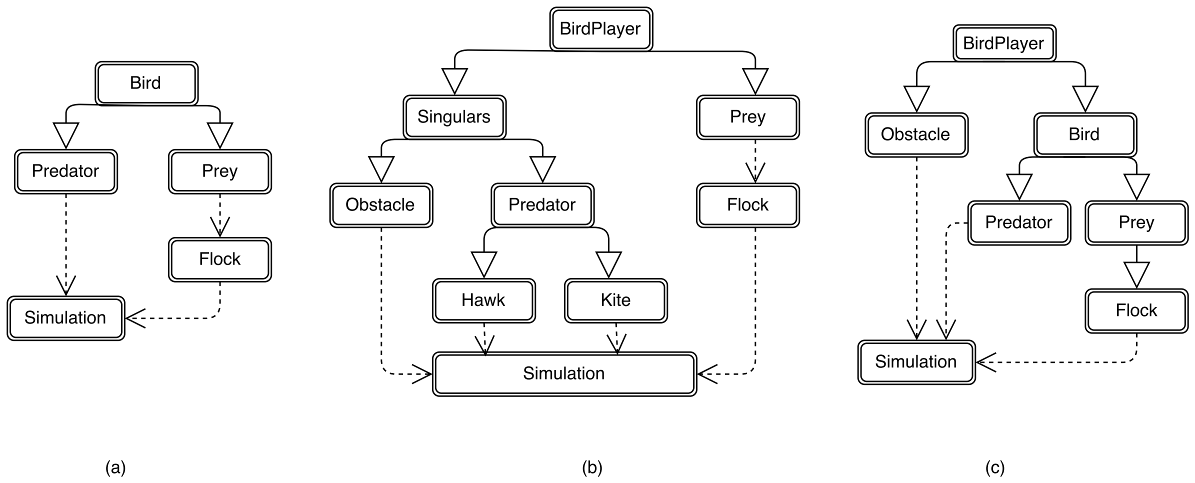

Figure 7: *Evolution of program structure leading towards design shown in figure (3). (a) The first design contains a* ***Flock*** *class, storing pointers to Prey objects and passing them to each other and the predator within* ***Simulation****. (b)* ***Singulars*** *are objects that do not flock. (c) Replacing Flock with AviationSystem and linking this to Obstacle and Predator gives the final code structure.*

6 Extensions to the Model

6.1 Blind Spots and Rotation

It is certainly possible to draw a cone for each object, adjusting its alignment when steering in the $x - y$ plane and adjusting its height to give a sense of perspective along the z-axis. A local angle of rotation could then be measured from the 'head' of the bird and a 'blind spot' centred directly opposite this angle. Each neighbour's direction with respect to this angle could then be defined, averaged over, and evolved towards.

6.2 Goal Seeking

A possible child of the Obstacle class might be an explicitly defined **Goal** class, whose motion could be specified to follow a particular path about the virtual space. Extending this, multiple paths might be defined, either within the same class or for children of this class. The flock could then follow different paths depending on which movement function the Goal calls, or which child of Goal it is told to $FlockWith$.

6.3 Prey, Predator and Obstacle as Base Classes

The three currently instantiable classes could become pure virtual, with real children inheriting data and functions from them. An interesting experiment could be to create a **Cloud** child of Obstacle, with movements designed to cause flock-like clustering, through which Birds would be able to move unimpeded. Additionally, different species of prey bird could exhibit a range of behaviours, whilst multiple predator species could hunt in a variety of ways. For example, a **Vulture** class could only hunt 'tired' birds with speed less than the flock average.

6.4 Aviary

This concept builds on that described above. Multiple species of bird will have the same defining Bird characteristics, but may have different sight ranges, speeds, flocking distances and other movement-

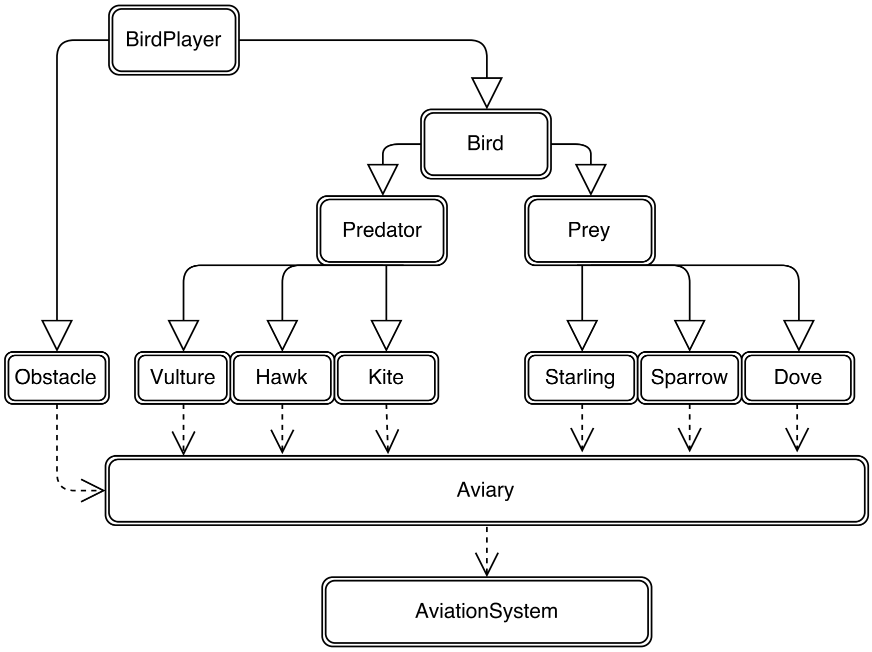

Figure 8: *Amalgamation of all classes inheriting from Bird into Aviary, which becomes a library of Bird-type classes.*

determining parameters. In such a system of multiple types of prey and predator, an extra header file could be used to *include* all child classes of Bird. This encapsulation of all Bird-type objects is demonstrated in figure (8). The AviationSystem class needs only to *include* Aviary in order to access all possible types of Predator and Prey.

7 Conclusion

Object-oriented programming techniques dramatically simplifies the simulation of bird motion by the *Birds* program, which uses alternative bird movement algorithms to those used in the well-known *Boids* flocking simulation. This produced an aesthetically pleasing display with interactive controls that enable user modification of the system. The model has many possible extensions, with future evolutions, featuring birds of multiple species with blind spots and alternate paths, under consideration and eagerly anticipated.

8 Sources

8.1 Included External Classes

(a) Sergi, A. *ThreeVector.* Written 25.09.2015. Accessed 26.11.2016 at *https://canvas.bham.ac.uk/courses/19528*, Exercise 11, Solution.

8.2 References and Further Reading

1. Reynolds, C. W. *Flocks, Herds and Schools: A Distributed Behavioural Model.* Published in *Computer Graphics*, 21(4), pp. 25-34, July 1987. Accessed 29.11.2016 at $http: //www.cs.toronto.edu/\ dt /siggraph97 - course/cwr87/$

2. Saraja, O. *Video example of Boids in action* [video], 28.12.2007. Accessed 05.12.2016 at $https : //www.youtube.com/watch?v = GUkjC - 69vaw$

3. Lewallen, R. *4 Major Principles of Object-Oriented Programming.* Published 19.07.2005. Accessed 02.12.2016 at $http : //codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/$

4. Barkakati, N. *Object-Oriented Programming in C++.* Stanford University, USA. Published 1991 by SAMS.

5. Johnsonbaugh, R. and Kalin, M. *Object-Oriented Programming in C++.* Depaul University, USA. Published 1995 by Prentice-Hall, Inc.

6. *Boids With A Fuzzy Way Of Thinking.* Faculty of Computer and Information Science, Slovenia. Published in *Proceedings of ASC*, pp 58-62, 2003. Accessed 02.12.2016 at $http : //itzsimpl.info/ papers/ilb_ASC03.pdf$

9 Appendix

B I R D P L A Y E R

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
BirdPlayer	PlayerType *Player*, ThreeVector *Mom*, ThreeVector *Pos*, double *MaxMom*	PlayerType defines enumerated list (*Neutral, Friendly, Dangerous*), declared *public* in BirdPlayer. Constructor initialises all private data members.

P R I V A T E M E M B E R S		
Identifier	**Data Type**	**Description**
fPlayerType	PlayerType	Player type of BirdPlayer
fMom	ThreeVector	BirdPlayer momentum
fPos	ThreeVector	BirdPlayer position
fMaxMom	double	Maximum BirdPlayer momentum

P U B L I C M E T H O D S			
Identifier	**Data Type**	**Parameters**	**Description**
RandPos	ThreeVector		Returns ThreeVector with random *x, y* components and maximum *z* scaled to virtual space
RandomVector	ThreeVector		Returns ThreeVector with random components between -0.5 and 5
GetPos	ThreeVector		Returns *fPos*
GetMom	ThreeVector		Returns *fMom*
GetMaxMom	double		Returns *fMaxMom*
SetPos	void	ThreeVector *Pos*	Assigns *fPos = Pos*
SetMom	void	ThreeVector *Mom*	Assigns *fMom = Mom*
SetPlayerType	void	PlayerType *Player*	Assigns *fPlayerType = Player*
SetMaxMom	void	double *MaxMom*	Assigns *fMaxMom = MaxMom*
RestrictMom	void		Constrains magnitude of *fMom* to *fMaxMom*
BoundaryConditions	void		Restricts
Move	virtual void	BirdPlayer * *ThisPlayer*, double *NPlayers*	Pure virtual method
GetPlayerType	virtual int		Pure virtual method
~BirdPlayer	virtual		Virtual destructor

B I R D

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
Bird	PlayerType *ThisPlayer*, ThreeVector *Mom*, ThreeVector *Pos*, double *MaxMom*, BirdType *ThisBird*	BirdType defines enumerated list (Null, Hawk, Starling) declared *public* in Bird.

P R I V A T E M E M B E R S		
Identifier	**Data Type**	**Description**
fBirdType	BirdType	Bird type of Bird
fSight	double	Sight range of Bird

P U B L I C M E T H O D S			
Identifier	**Data Type**	**Parameters**	**Description**
GetSight	double		Returns *fSight*
SetSight	void	double *Sight*	Assigns *fSight = Sight*
UpdateAndRestrict	void	ThreeVector *delMom*	Increments Bird momentum by delMom and updates position accordingly, restricting momentum and position within predefined limits
RandomMovement	void	double *NPlayers*	Adds randomness to Bird motion, scaling momentum change inversely to number of times method used within one timestep, *NPlayers*
Avoid	void	ThreeVector *Position*, double *NPlayers*	Steer away from *Position*with weighted momentum change
GetBirdType	virtual int		Pure virtual method
~Bird	virtual		Pure virtual method

OBSTACLE

CONSTRUCTOR			
Identifier	**Parameters**	**Description**	
Obstacle	ThreeVector *Mom*, ThreeVector *Pos*, double *MaxMom*	Calls BirdPlayer constructor without modification	
PUB LIC METHODS			
Identifier	**Data Type**	**Parameters**	**Description**
GetPlayerType	virtual int		Returns *Neutral*
Move	virtual void		Evolves position of obstacle over one time step
~Obstacle	virtual		Virtual destructor

PREY

CONSTRUCTOR			
Identifier	**Parameters**	**Description**	
Prey	ThreeVector *Mom*, ThreeVector *Pos*, double *MaxMom*	Calls the Bird constructor and assigns a sight range specific to Prey	
PUB LIC METHODS			
Identifier	**Data Type**	**Parameters**	**Description**
FleeFrom	void	ThreeVector *PredPos*, double *NPlayers*	Causes weighted momentum change away from *PredPos* if within the Prey sight range
FlockWith	void	ThreeVector *FriendPos*, double *NPlayers*	Causes weighted momentum change to optimise distance to *FriendPos* if within Prey sight range
GetPlayerType	virtual int		Returns *Friendly*
GetBirdType	virtual int		Returns *Starling*
Move	virtual void	BirdPlayer * *ThisPlayer*, double *NPlayers*	Evolves Prey position with respect to BirdPlayer referenced by pointer *ThisPlayer*
~Prey	virtual		Virtual destructor

P R E D A T O R

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
Predator	ThreeVector *Mom*, ThreeVector *Pos*, double *MaxMom*	Calls the Bird constructor and assigns a sight range specific to Predator

P U B L I C M E T H O D S			
Identifier	**Data Type**	**Parameters**	**Description**
Hunt	void	ThreeVector *PreyPos*	Causes momentum change towards *PreyPos* if within Predator sight range
GetPlayerType	virtual int		Returns *Dangerous*
GetBirdType	virtual int		Returns *Hawk*
Move	virtual void	BirdPlayer * *ThisPlayer,* double *NPlayers*	Evolves Predator position with respect to BirdPlayer referenced by pointer *ThisPlayer*
~Predator	virtual		Virtual destructor

A V I A T I O N   S Y S T E M

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
AviationSystem	int *NPrey*, double *PreyMaxMom*, int *NPred*, double *PredMaxMom*, int *NObs*, double *ObsMaxMom*	Initialises the AviationSystem with a specific number of objects of given type and their corresponding maximum momenta

P R I V A T E M E M B E R S		
Identifier	**Data Type**	**Description**
fPreyFlock	std::vector< Prey* >	Vector of pointers to Prey objects
fPredFlock	std::vector< Predator* >	Vector of pointers to Predator objects
fObsFlock	std::vector< Obstacle* >	Vector of pointers to Obstacle objects
fAllPlayers	std::vector< BirdPlayer* >	Vector of pointers to objects inheriting from virtual base class BirdPlayer
fGoal	Obstacle	Separate new instance of Obstacle for Prey to seek

PUB LIC METHODS			
Identifier	**Data Type**	**Parameters**	**Description**
**GetPrey*	Prey	int *ptr*	Returns component *ptr* of *fPreyFlock*
**GetPredator*	Predator	int *ptr*	Returns component *ptr* of *fPredFlock*
**GetObstacle*	Obstacle	int *ptr*	Returns component *ptr* of *fObsFlock*
PreyNumber	int		Returns size of *fPreyFlock*
PredNumber	int		Returns size of *fPredFlock*
ObsNumber	int		Returns size of *fObsFlock*
AddPrey	void	Prey * *newprey*	Pushes pointer *newprey* into *fPreyFlock* vector
AddPred	void	Predator * *newpred*	Pushes pointer *newpred* into *fPredFlock* vector
AddObs	void	Obstacle * *newobs*	Pushes pointer *newobs*into *fObsFlock* vector
Simulate	void		Evolves the position of every object within the system by one time step
ErasePrey	void	int *ptr*	Releases memory storing component *ptr* of *fPreyFlock* and removes pointer from vector
ErasePred	void	int *ptr*	Releases memory storing component *ptr* of *fPredFlock* and removes pointer from vector
EraseObs	void	int *ptr*	Releases memory storing component *ptr* of *fObsFlock* and removes pointer from vector
~AviationSystem			Real destructor releasing all memory used to store pointers and erasing all corresponding vector pointers

D I S P L A Y   W I N D O W

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
DisplayWindow	AviationSystem *system, Qwidget *parent	Constructor sets initial appearance of window and defines rate of animation

P R I V A T E M E M B E R S		
Identifier	**Data Type**	**Description**
*ui	Ui::DisplayWindow	Pointer to DisplayWindow.ui form
*fSystem	AviationSystem	Pointer to current system under simulation
fModelSystem	std::vector< BirdPlayer* >	Vector of pointers to objects inheriting from virtual base class BirdPlayer
fOpacity	bool	Boolean value storing viewer opacity decision

P U B L I C M E T H O D S			
Identifier	**Data Type**	**Parameters**	**Description**
PUBLIC			
~DisplayWindow			Real destructor deleting pointer *ui*
PRIVATE SLOTS			
paintEvent	void	QPaintEvent *event	Draws current position of each visible object within the system to the screen, updating upon *timeout* signal sent by 10ms timer
on_ThreeD_toggled	void	bool *checked*	Sets *fOpacity* true if checked and false otherwise, called upon ticking or unticking the check-box on the window

M A I N   W I N D O W

C O N S T R U C T O R		
Identifier	**Parameters**	**Description**
MainWindow	Qwidget **parent*	Constructor sets initial appearance of window, defines rate of system evolution and initialises new AviationSystem object to interact with
P R I V A T E M E M B E R S		
Identifier	**Data Type**	**Description**
**ui*	Ui::MainWindow	Pointer to MainWindow.ui form
fRunStatus	int	Integer of value 0 or 1, corresponding to items *Running* or *Idle* in privately enumerated list, specifying current simulation state
**fAviationSystem*	AviationSystem	Pointer to system under simulation
**fCurrentWindow*	DisplayWindow	Pointer to current display window
fPreyMaxMom	const double	Constant maximum Prey momentum
fPredMaxMom	const double	Constant maximum Predator momentum
fObsMaxMom	const double	Constant maximum Obstacle momentum

PUB LIC METHODS			
Identifier	**Data Type**	**Parameters**	**Description**
PUBLIC			
~MainWindow			Real destructor deleting pointer *ui*
PRIVATE SLOTS			
Animate	void		Evolves fAviationSystem over one time step if simulaiton is running, called upon timeout signal sent by 10ms timer
on_Run_clicked	void		Called by click of *Run* button, function updates *fRunStatus,* creates new Display Window and changes appearance to red *Stop* button if simulation *Idle*. When simulation *Running,* clicking *Stop* closes the display.
on_AddPrey_clicked	void		Clicking *Add Prey* button provokes new instance of Prey, whose pointer is inserted into the *fAviationSystem* Prey pointer vector
on_AddPrey_clicked	void		Clicking *Add Predator* button provokes new instance of Predator, whose pointer is inserted into the *fAviationSystem* Predator pointer vector
on_AddObs_clicked	void		Clicking *Add Obstacle* button provokes new instance of Obstacle, whose pointer is inserted into the *fAviationSystem* Obstacle pointer vector
on_DelPrey_clicked	void		Clicking *Delete Prey* button removes last Prey object from *fAviationSystem*
on_DelPred_clicked	void		Clicking *Delete Predator* button removes last Predator object from *fAviationSystem*
on_DelObs_clicked	void		Clicking *Delete Obstacle* button both removes last Obstacle object from *fAviationSystem* and updates range of the Obstacle selection spin-box to current number of visible Obstacles within *fAviationSystem*
on_ObsMomX_sliderMoved	void		Moving x-momentum slider modifies x-axis momentum of Obstacle referenced by spin-box
on_ObsMomY_sliderMoved	void		Moving y-momentum slider modifies y-axis momentum of Obstacle referenced by spin-box
on_ObsMomZ_sliderMoved	void		Moving z-momentum slider modifies z-axis momentum of Obstacle referenced by spin-box