Fleck: An OpenGL Flocking System

Shane Aherne
MSc Computer Animation
NCCA, Bournemouth University

Computer Animation Techniques

Technical report

Abstract

To implement a Flocking System based on Reynolds' original paper. A moving goal will be specified by the user, or can be interactively adjusted by the user. The flocking system should take into account the environment, and other factors such as predator-prey relationships.

1.0 Project Introduction

Flocking systems have extended the possibilities of modern movie making to a large extent over the past few years. In films such as *Lord of the Rings [5]*, *Troy [7]* and *Kingdom of Heaven [6]*, huge epic battle scenes have been made possible by implementing either in-house, or commercial software. Probably the most popular commercial software is Massive, a crowd simulation system implementing individualized behavior, energies and interactions. Battle scenes aside, Massive has been utilized to create the award winning *Playstation 2 Mountain Ad*, and *Nike's "Euro 2004 - The Other Game" ,* both produced by *The Mill* in London.

The aim of this project is to produce an application which will act as a foundation for future work. Basing the application on 'Flocks, herds and schools: A distributed behavioral model', [1] a simple school of fish model will be simulated.

2. 0 Approach Adopted

This Flocking System is based on the rules presented by Reynolds in 'Flocks, herds and schools: A distributed behavioral model' [1] . The boids within this application base their navigation module on the principles of separation, alignment and cohesion.

They share a common goal, and avoid obstacles in their path. The boids also react to a predator, who targets in on their central average.

Throughout this report, each method is broken down. The final system user interface is also explained. The graphical user interface allows a user to interact with the flock, changing its leader path, the existence of predators, and adjust its centering and leader vector strengths.

3. 0 System Overview

The system front end is creating using QT. The program is written in C++, utilizing OpenGL. Jon Macey's GraphicsLib library is also used.

4.0 Behaviour Breakdown

4.1 Neighbouring with closest Boids

The computation used in this application, has N^2 complexity. Whereby, each boid checks every other boid to determine its closest neighbours. There are several other methods for achieving this.

In some cases a Bucket Sort algorithm is used, where adjacent buckets are check for neighbours . This involves a time space trade-off depending on bucket size, therefore the smaller the buckets the more buckets needed, but on average fewer members per bucket.

Other methods include message passing, where each boid informs others of its position.

This system is set up for a region parenting lattice. Whereby, the area the boids travel in is gridded out into a sequence of grids/bounding regions. Each boid in turn is a neighbor of other boids within that *Axis Aligned Bounding Box.* When a Boid moves between bounding boxes, its parent is updated.

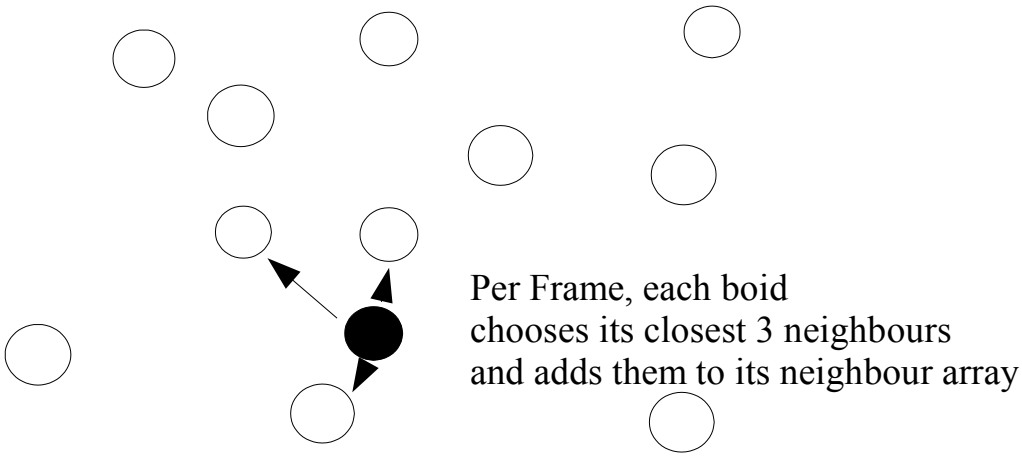

Figure 4.1 – Neighbouring with closest boids

When a Boid's Centering, Velocity and Separation Vectors are called, it only checks data of its closest neighbours. This gives a far more natural look and feel. It is also more computationally efficient.

4.2 Centering

Each boid has an urge to centre with nearby flock mates. As a boid only has knowledge about nearby boids, a far more natural look and feel exists. In fact, almost simulating, a model of a school of fish in murky water.

If a boid is on the boundary of the flock, the urge to centre is stronger than that of a boid who is already approximately at the centre.

The centering is quite easily calculated, by going through the array of neighboring boids, and averaging their positions. This centre vector is added to the final navigation module.

4.3 Separation

4.3.1 *Avoiding Other Agents*

The agents must avoid colliding with each other to maintain the aesthetic look of a natural flock.

The method adopted was based around the Linear Collision response method proposed by Bourg, 2002 [3]. This provided quite a visually acceptable collision response algorithm.

2 factors are addressed in this method:
- Whether or not the objects are close enough, within numerical tolerances, to be considered in colliding contact.
- What the relative normal velocity is between the objects

As the boids have bounding spheres, the center points provide the centre of mass where all measurements are calculated from.

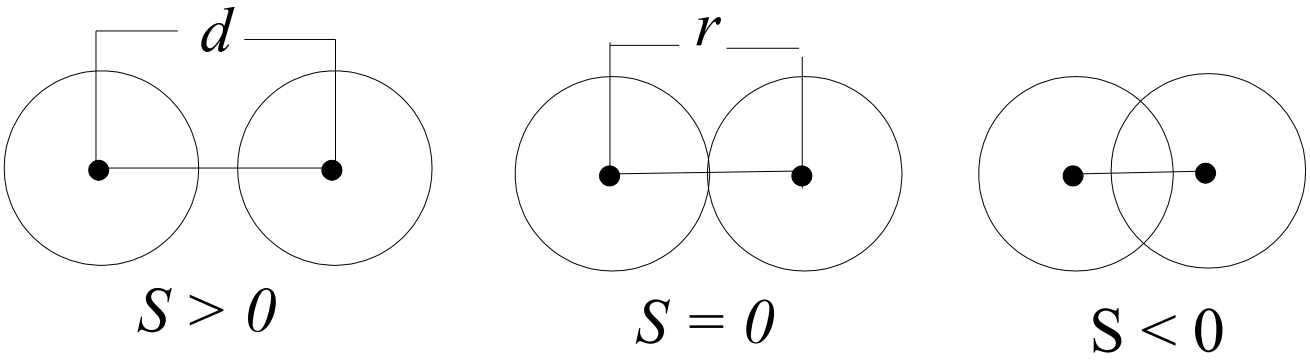

Fig 4.3.1 - Bounding Sphere Collision States

Initially, a vector between the 2 centres is calculated by subtracting the second bounding sphere centre from the first. Providing us with vector *d* .The value s is found by calculating the length of d and subtracting the sum of the two radii of the bounding spheres. If $s > 0$, then no collision has occurred. If $s = 0$, then the two spheres are colliding. If $s < 0$, the two spheres are penetrating each other.

Vector d is now used to calculate the "collision normal" by normalizing the value. At this stage the velocities of the agents are checked to determine if they are moving towards each other. This is achieved by subtracting the velocity of the first agent from the second agent, then taking the dot product of the resulting value and the collision normal. If this resulting value is negative then the agents are moving toward each other, otherwise it is determined that the agents are moving away from each other.

If it has been determined that the boids are approaching a collision then the collision response algorithm is invoked.

This collision response method is a relatively simple function which uses the *vector s* to repel the boids away from each other. This is weighted to give smoother movement.

4.4 Velocity Matching

Each boid calculates an average velocity with its neighbours. As discussed in Reynolds [1] paper, a minimum and maximum velocity exist. Once an agent breaks this range, the velocity is automatically adjusted to maintain a smoother movement.

Velocity matching and collision avoidance with nearby flock mates are complementary. When used correctly together they help each boid in the flock to be free flowing.

4.5 Goal Setting

The boids share a common goal. For each update call, the position of the leader is updated. Each boid is made aware of the leaders position and creates a goal vector based on this information. The goal vector is calculated by simply subtracting the leaders position from the boids position.

This vector is then normalized, and added to the navigation module.

4.6 Collision Avoidance

Avoiding Environmental Obstacles

A flock can become rather sterile looking once isolated from an environment. By adding objects into the scene, a far more interesting motion begins to emerge.

To detect an impending collision, a ray-sphere collision detection algorithm is utilized. Following this the boid steers toward the boundary of the obstacles bounding sphere. The algorithms involved are explained here.

4.6.1 *Ray Sphere Collision Detection*

Ray Sphere Function

Calculates if the current vector of the boid is in a collision path with an object in the scene.

The points of a sphere center Pc(x, y & z) are passed to this function, along with the sphere radius.

The equation of a sphere centered at P_c is $(P – P_c) . (P – P_c) – r^2 = 0$

The ray equation being $p = t_d + P_o$

Combining this with the sphere equation : $(t_d + Po – P_c) . (t_d + Po – Pc) - r^2$
Create a quadratic equation from this: $t^2 + bt + c = 0$

Calculate the distributive property of the dot product:
$a = d . d$
$b = 2 * d . (Po – Pc)$
$c = (Po – Pc) . (Po – Pc) - r^2$
$D = b^2 – 4 (a * c)$

So, if D < 0.0 there's no hit. If D = 0, the ray is on tangent to the sphere, and if D is positive a collision will exist.

If D is positive the Boid must calculate a new path to avoid collision

4.6.2 *Steering to the boundary point on the bounding sphere*

Based on calculating a point on the boundary of a sphere, presented on page 254 of Rick Parents book. [2]

Collision Avoidance function
If the Boids vector displays collision with a bounding sphere, this function tests to see if the boid is within a sphere of influence of the Obstacle. If the boids is within this distance, it must calculate a new direction vector to avoid collision with the object:

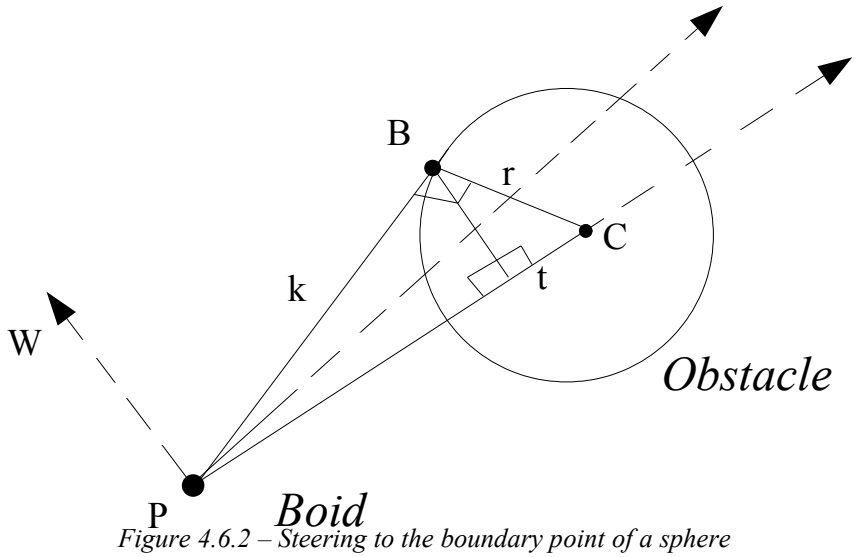

Figure 4.6.2 – Steering to the boundary point of a sphere

B is calculated using the following Formula:

$$B = P + (| C - P | - t) \, . \, U + s \, . \, W$$

where $U = C - P \, / \, | C - P |$

$W = (U \times V) \times U \, / \, | (U \times V) \times U |$

$s = \sqrt{r^2 - t^2}$

Once the Collision Avoid Boolean value is set to true (when the Boid is within a certain region of the Obstacle), the collisionAvoid vector is added heavily to the overall boid vector. This provide sufficient Obstacle Collision capabilities

4.7 Spherical Rotation

Rotational Position of the Boids

When the boid is at the origin, and needing to rotate its position towards the point *U*, the following formula is adapted from *Virtual Camera [9]* maths.

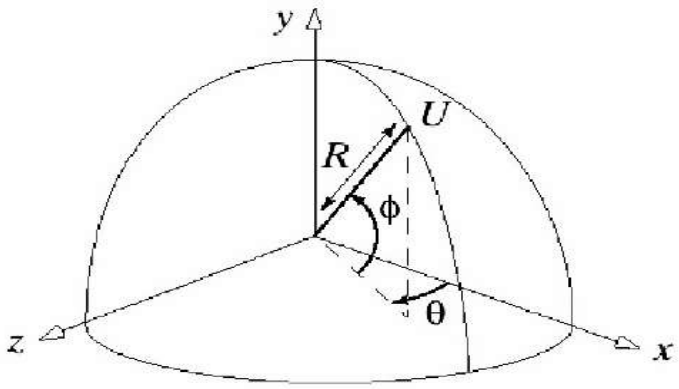

Figure 4.7 – Rotational Poisitions of the boids

R is the radial distance of ***U*** from the origin.
Ø is the angle ***U*** makes with the ***xz-***plane (Latitude of the point ***U***)
Θ is the azimuth of ***U*** (The angle between the ***xy –*** plane and the plane through ***U*** and the *y-axis*)

Ø is calculated using the following formula
Ø= sin^{-1} (***uy/R***)

Θ is calculated using the following formula
Θ = arctan (*uz , ux*)

In C++ sin^{-1} can be called by *asin,* while *arctan* is called by *atan2*

Within the application the function *calcPhiTetha()* is used to calculate these values. The ***x, y*** and ***z*** values are the difference in the ***x,y*** and ***z*** coordinate between the current and previous position. Therefore, creating a rotation based on the previous position of the boid and the aim vector.

4.8 Predator/Prey Relationship

4.8.1 Predator

The Predator Class is actually quite basic yet effective. The predator begins at a random position, with a random direction vector.

A new direction Vector is immediately applied to this predator from the flock class. This direction vector, points towards the centre of the flock.

Per frame, the Predator's update target function is called. Within this function the Distance to the center of the flock is measured using Pythagoras 's theorem.

If the distance is less than a certain influence region, the boolean value target reached is called. Once this is set to true, the predator decreases its velocity by a certain factor per frame.

If the target reached value is still set to false (i.e. The distance is greater than the region of influence), then the predator increases its velocity by a certain factor per frame.

Once the Predator's velocity is below a certain minimum cap, then a new target Vector is calculated (from the flock centre point at that stage), and the target reached boolean value is set to ***false.***

The result of this is that a predator locks onto a centre flock region, constantly increases its speed on the approach, and once through the centre slides to a halt, and assesses where to target next.

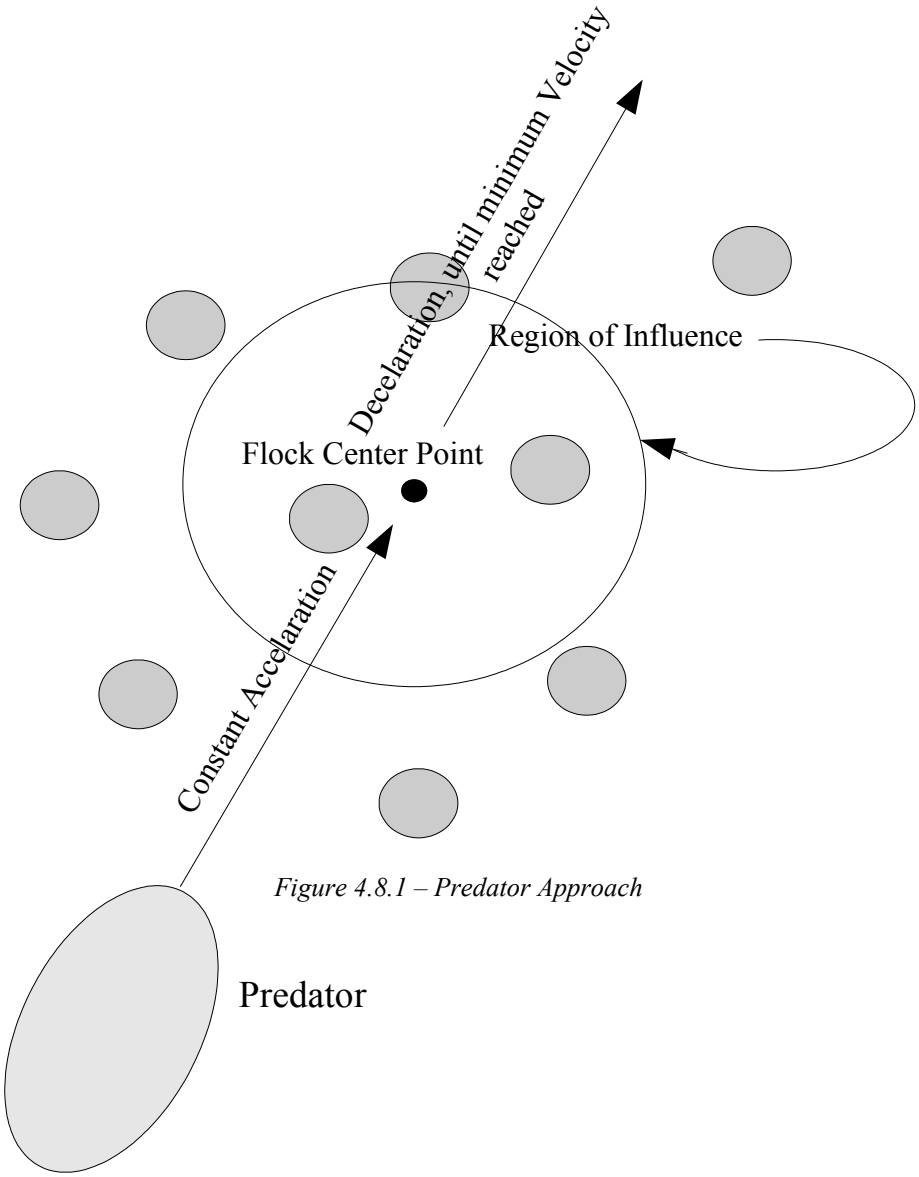

Figure 4.8.1 – Predator Approach

4.8.2 Prey

Within this application, the Boids are always aware of the position of the predator.

Once the predator has broken a certain bounding region of the boid, the Vector between the Predator and the Boid is calculated.

This Vector is used to push the boid away from the predator until it is outside the danger region.

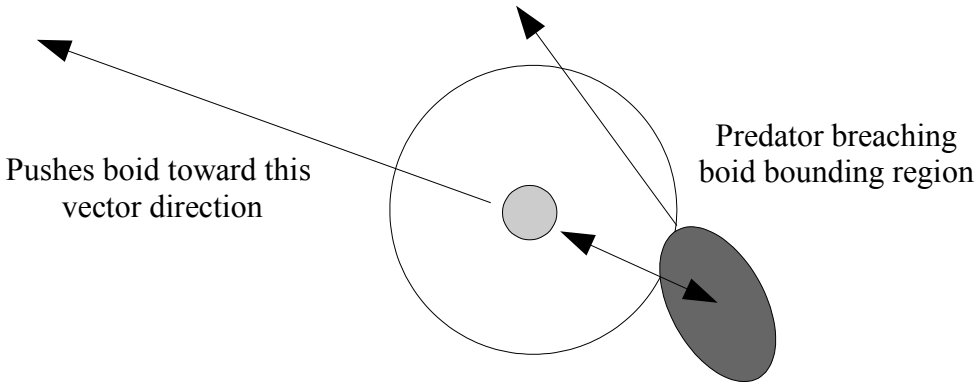

Figure 4.8.2 - Prey Reaction

The overall result gives quite a realistic Prey/Predator relationship, while maintaining a degree of computational efficiency to the program.

4.9 Bounding Box Region

An *Axis Aligned Bounding Box* is utilized in this system, to specify a bounding region for the leader boid. As a result, the leader boid will maintain a path until a certain coordinate minimum or maximum value is reached by the objects ray.

Once this has occurred the leader boid will change its direction. This change in direction is weighted depending on the distance to the bounds.

4.10 The Flight Module

The Vectors calculated from the previous equations are combined (with different weighting, and prioritizations) to create the *Navigation Module*. Following this, the constraints and status of the Boids are applied to form the *Pilot Module*. This *Pilot Module* in turn passes its result to the *Flight Module*, which attempts to fly in that direction.

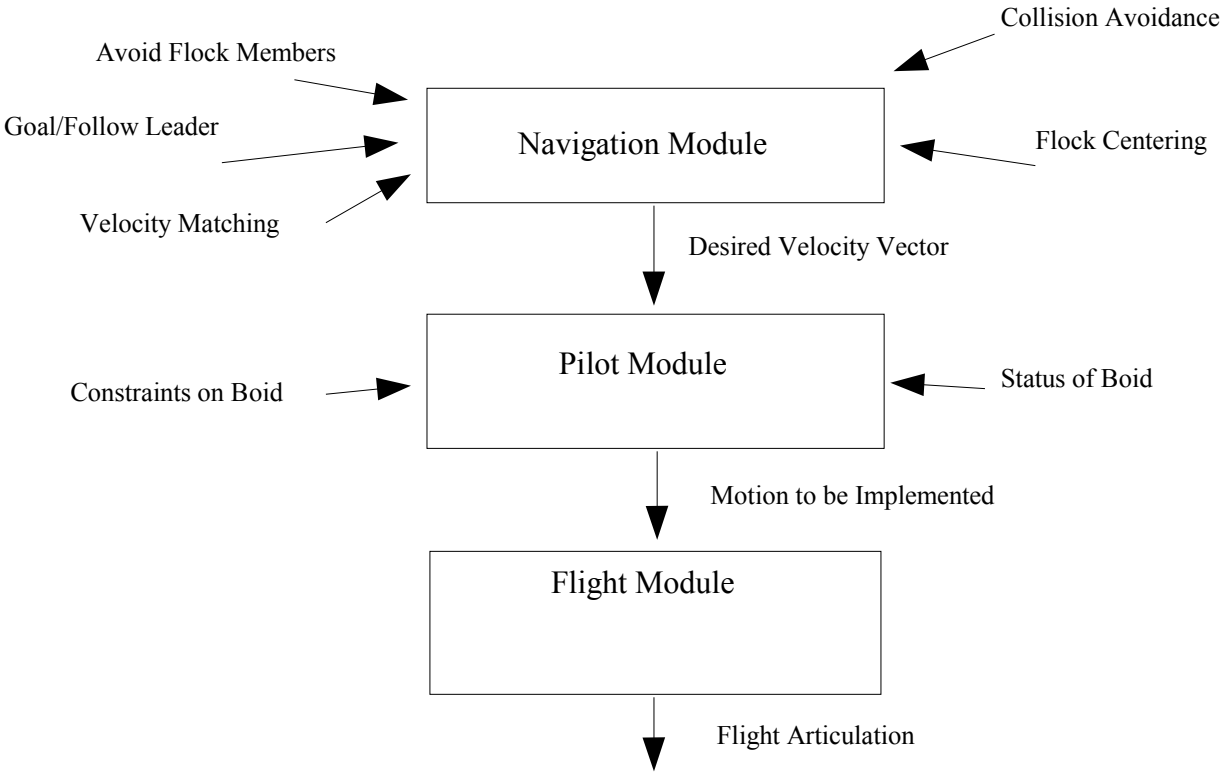

Figure 4.10 – Negotiating the Motion

5.0 Tool Overview

5.1 User Interface

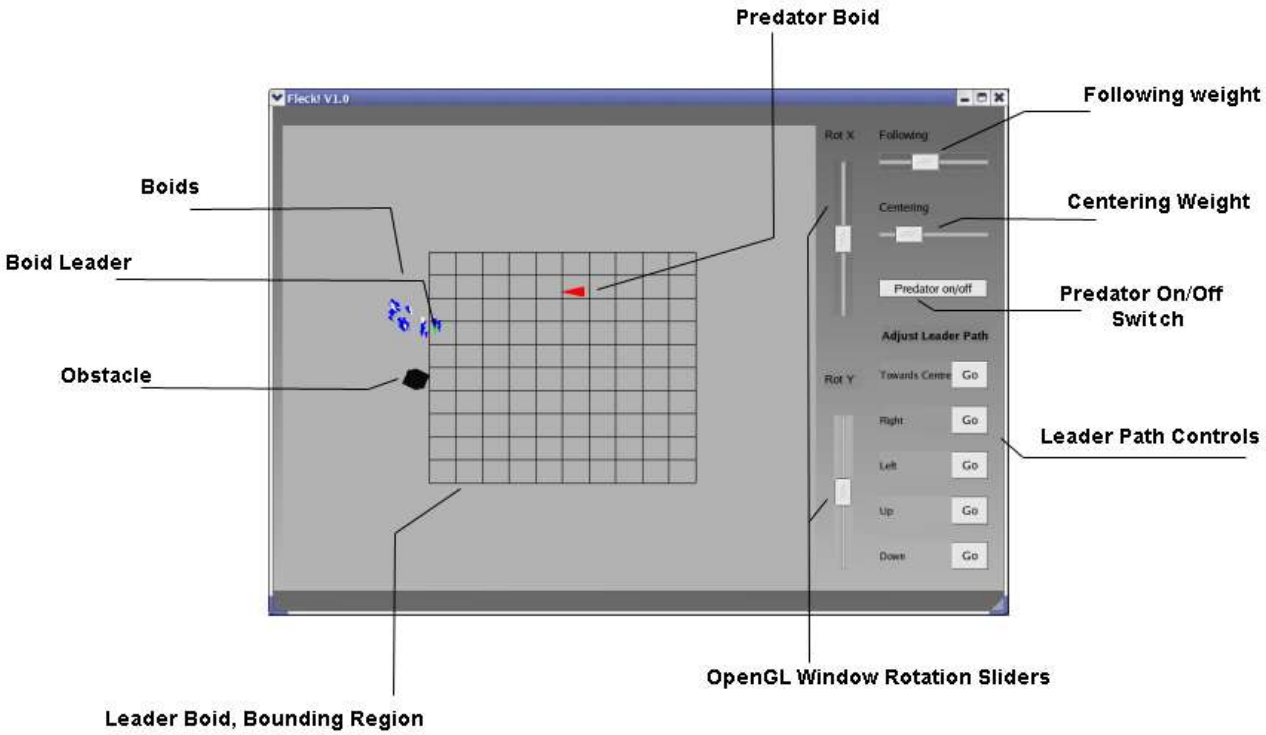

Figure 5.1 – Overview of User Interface

The User Interface allows a certain amount of interaction with the flock.

Following Weight: Using the slider the user can adjust the weight of the goal vector in navigation module. Placing the slider to its lowest value gives quite nice looking results.

Centering Weight: The user can increase the urge of the boid to centre with its closest flock mates.

Predator On/Off: Allows the user to activate the predator attack, or disable the attack urge.

Adjust Leader Path: By clicking these buttons, it is possible to navigate the leader boid around the bounding box region.

OpenGL Window Rotation Sliders: The user can rotate around the 3D scene by adjusting these sliders.

5.2 Class Diagram

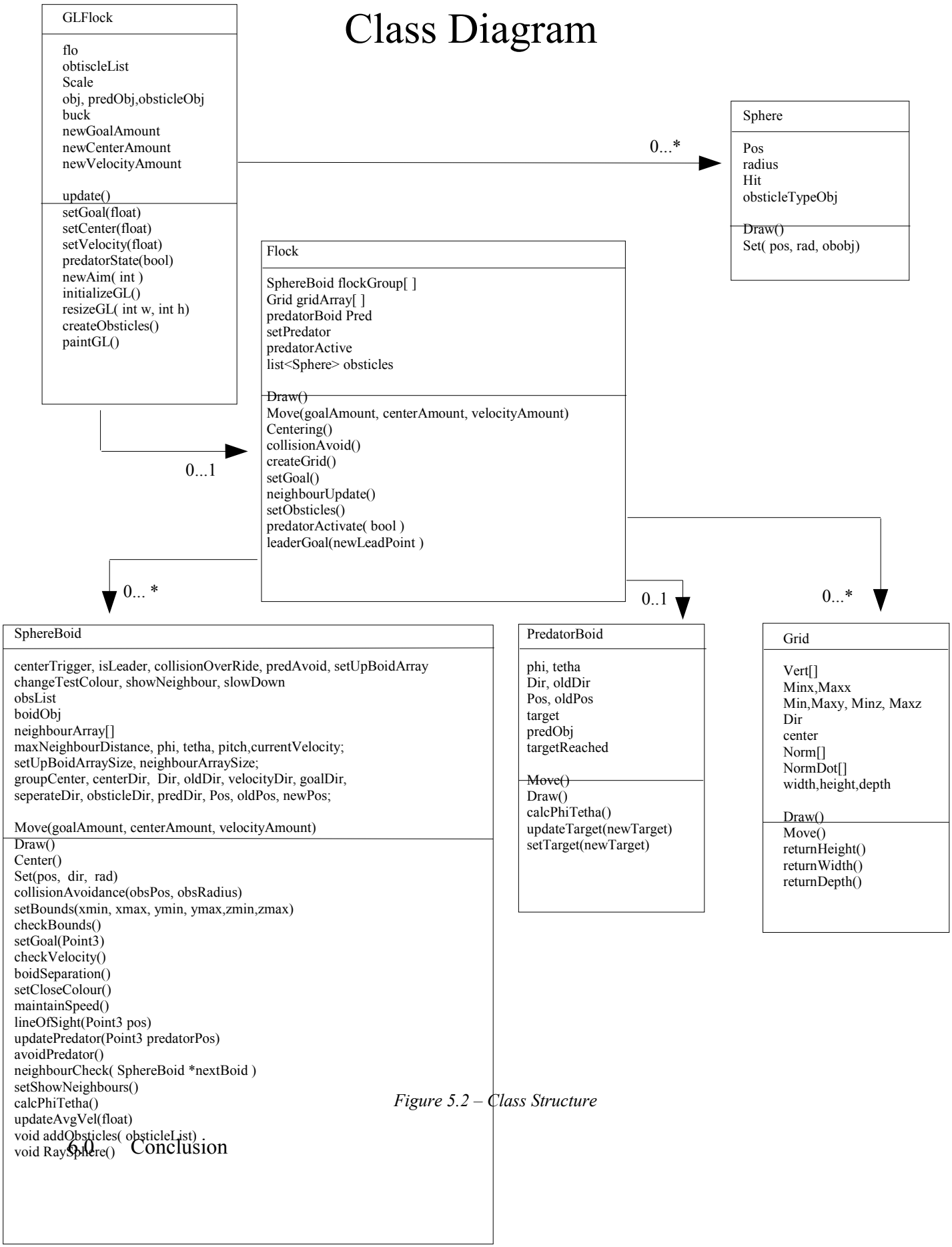

Figure 5.2 – Class Structure

6.0 Conclusion

Overall this application has proved a success, with plenty of scope for extensions. The user interface presented a more efficient method of achieving a desired look and feel. Future work could include a more interactive graphical user interface allowing a range of different flocking styles to be created quickly.

Further work, could also concentrate on achieve bucket sorting for neighbour relations, or extending the grid system presented here. More intuitive path planning and separation procedures also present room for improvement.

Acknowledgments

I would like to thank Jon Macey and Prof. Jian Jun Zhang of the NCCA, Bournemouth University. I would also like to thank Lucy Ward, Osiris Perez and David Basalla.

References

[1] Craig W. Reynolds, Flocks, herds and schools: A distributed behavioral model, ACM SIGGRAPH Computer Graphics, v.21 n.4, p.25-34, July 1987

[2] Parent R, 2002.Computer Animation Algorithms and Techniques, Morgan Kaufmann Publishers, San Francisco, USA

[3] Bourg, D . 2002. Physics for Game Developers. O'Reilly and Associates

[4] Gino van den Bergen, Efficient collision detection of complex deformable models using AABB trees, Journal of Graphics Tools, v.2 n.4, p.1-13, April 1997

[5] Lord of the Rings, 2001. Film. Directed by Peter Jackson. USA New Line Cinema

[6] Kingdom of Heaven, 2005. Film. Directed by Ridley Scott. USA, 2Oth Century Fox

[7] Troy, 2004. Film. Directed by Wolfgang Peterson. USA, Warner Brothers

[8] OB Bayazit, JM Lien, NM Amato . Simulating Flocking Behaviors in Complex Enviroments, Pacific Conf. on Computer Graphics and Applications, 2002

[9] Macey J, 2004, The Virtual Camera, Retrved on 09/04/05 from
http://dec.bournemouth.ac.uk/staff/jmacey/proggraph/VirtualCamera.pdf

[10] Macey J, 2004, Collision Detection, Retrieved on 20/04/05 from
http://dec.bournemouth.ac.uk/staff/jmacey/proggraph/Collisions.pdf,

[11] Vince, J. 2001, Essential Mathematics for Computer Graphics Fast, Springer

[12] Oulline, S. 1995. Practical C++ Programming. O'Reilly & Associates

[13] Crombie, D, 2000, Rules for Flocking, Retieved on 03/05/05 from
http://members.ozemail.com.au/~dcrombie/project/swarming.html

[14] Robbins J, 1997, The Boids, Retrieved on 10/05/05 from
http://www.geocities.com/SiliconValley/Vista/1069/Boid.html

[16] Kline C, 1996, C++ Boids, Retrieved on 10/05/05 from
http://www.behaviorworks.com/people/ckline/cornellwww/boid/boids.html