Final Report – Contents

1. Purpose of this Document

This document describes the overall purpose of the project, outlines past work involving computer-simulated flocking, and gives a chronological listing of the project's development. A list of references used throughout the design and developmental cycles is also included.

2. Abstract

To many, a flock of birds is one of nature's beautiful sites. The patterns of the motion seem choreographed and complex. However, it is believed that the flock's movement is not guided by a single "leader bird." Rather, it is the combination of each bird's relatively simple instincts which create the aggregate motion. The potential of this decentralized ideology is becoming evermore apparent for Computer Science applications.

It is the aim of this project to create flock animations by simulating the behaviours of the individual birds (actors). This model is an elaboration of a "particle system"– each actor has its own properties, such as position and velocity, and reacts independently according to its local perceptions of the universe.

Furthermore, this project is implemented as a "plug-in" custom effect for the popular Softimage|3D animation system. It is hoped that, with the combination of a behavioural simulation of the flock and a powerful modeling/rendering system, convincing animations of flock motion can be generated.

3. Analysis of previous works

3.1 Particle Systems
Allen Martin

Particle systems are a technique used to graphically model and/or animate certain applications. Although the term is used loosely, there exist two conditions which hold true for all particle systems:

1. " Collection of particles – A particle system is composed of one or more individual particles. Each of these particles has attributes that directly or indirectly effect the behavior of the particle or ultimately how and where the particle is rendered. Often, particles are graphical primitives such as points or lines, but they are not limited to this…
2. Stochastically defined attributes – The other common characteristic of all particle systems is the introduction of some type of random element. This random element can be used to control the particle attributes such as position, velocity and color. Usually the random element is controlled by some type of predefined stochastic limits, such as bounds, variance, or type of distribution. "

In this project, each particle is a member of the flock. Throughout this project's documentation, the term *actor* is used to denote a member of the flock. Each actor has a number of attributes such as its position and velocity. The "random element" mentioned above is introduced in this project by means of each actor's behaviours, or *instincts*.

The actor's instincts influence its attributes. For example, the actor's instinct to avoid colliding with obstacles will cause its position and velocity to steer away from nearby objects.

Particle systems have been used in Computer Science to model things such as: fire and explosions, trees, smoke and, yes, even flocking birds (see description of Reynolds' work below).

More sophisticated particle systems allow the particles to be born, grow old and die after a certain period of time. This project uses a simpler, more-general model which does not require these features.

3.2 Flocks, Herds, and Schools: A Distributed Behavioral Model Craig W. Reynolds

This paper was first presented at the 1987 ACM *Special Interest Group on computer GRAPHics* (SIGGRAPH) conference. It has become the founding document for, specifically, flocking and distributed behavioural animation techniques, in general.

The paper aimed to produce an accurate simulation of the motion of a flock of birds in nature. Reynolds pointed out that the complex pattern of motion seen in the flock was the result of the interaction of relatively simple behaviours of each individual member. Each bird reacting upon its own instincts and perceptions creates the aggregate motion of the flock.

Reynolds modeled the flock after a particle system. "Particle systems are collections of large numbers of individual particles, each having its own behaviour." Each member of the flock has attributes, such as its position and speed, and behaviours based on its local perception of the universe.

His work came to be know as *Boids* because of the "bird-oid" objects being flocked. Its foundation was a sophisticated geometric and physical model. Factors such as bank, pitch, roll, acceleration, gravity and momentum were considered and taken into account.

At the heart (or mind) of each boid lay three main behaviours. These three rules guided the flight of each boid and were ultimately responsible for the motion of the flock.

> " 1. Collision Avoidance: avoid collisions with nearby flockmates
> 2. Velocity Matching: attempt to match velocity with nearby flockmates
> 3. Flock Centering: attempt to stay close to nearby flockmates "

Each of these behavioural urges produced a suggestion as to which direction the boid should steer. Reynolds also discussed behaviours to: add randomness to the flock, allow the flock to be controlled or scripted by the animator, and to avoid obstacles placed within the universe.

The results of each behaviour were combined in order of a specified priority until the magnitude of the result reached a given maximum.

Of course, the entire basis of this project follows in the footsteps of Reynolds' work. Many concepts and ideas were borrowed for this project, including:

- distributed behavioural model – a number of independent actors acting upon programmed urges and their local perceptions; adaptation of a particle system
- the idea of prioritizing and averaging the results of each behaviour, although implemented somewhat differently
- the ability to limit the actors' perceptions of the world; the concept of *nearby* flockmates
- ideas of behaviours for avoiding obstacles, following a scripted path and wandering
- scaling distance vectors by the inverse square of their magnitude instead of a, less accurate, linear approximation

Important differences between Reynolds' Boids and the project presented here are:

- lack of a sophisticated simulation of the laws of physics and motion
- Reynolds' original three behaviours for flocking (Collision Avoidance, Velocity Matching and Flock Centering) have been coalesced into a single behaviour which incorporates all of these urges
- greater control for the user to choose which actors are considered *nearby flockmates* for the flocking instinct
- specific implementation for an already established animation system (Softimage); also provides flexibility via a graphical user interface
- user has full control over priorities and options of each behaviour
- an instinct has been added to keep the members of the flock inside of a specific region of space (for control)

Reynolds' conclusions point out that "… it is difficult to objectively measure how valid these simulation [results] are." The same could be said of this project: Success is truly in the eye of the beholder.

3.3 Flocking Birds
John Mee

In 1994, a student at Curtin University in Australia wrote a flocking program as a project for his Post Graduate Diploma in Computing.

John Mee's work was also based on Craig Reynolds' *Boids* and was implemented using GL on Silicon Graphics workstations. His boids based their course decisions on seven criteria:

- continue – continue along previous course
- peers – fly in same direction as nearby flockmates, at the same speed
- collide – don't crash into flockmates
- altitude – maintain a comfortable height above the ground
- stray – after a period of time of being away from the flock, aim the boid back towards the flock
- speed – stay within valid speed boundaries
- whim – fly in a random direction
- avoiding objects – avoid collisions with obstacles

To summarize Mee's project: The movement of the flock for each frame was computed in two stages. The first calculated the priority for each criteria while the second actually computed the desired direction and speed. The results of each criteria were scaled, sorted by priority, and combined so as to not exceed a specified limit. The inclusion of a *panic mode* allows critical matters to be attended to at a priority much greater than normal.

Some key points of Mee's project include:

- A sophisticated collision avoidance implementation – a ray-casting method is used to determine possible collisions with obstacles and to plot a safer course
- An interactive, albeit somewhat confusing, graphical user interface
- Panic mode to provide "extra power to the really important decisions"

4. Project Development History

The following provides a chronological listing of the project's development. Before proceeding, however, it is useful to, at least, understand the general nature of the project's design. A brief summary of how the project functions is provided:

The flocking engine contains the main loop of the project. This loop iterates once for each frame of the resultant animation. During each frame, each actor is consulted for a new position and velocity. The actor's decision is based upon six programmed instincts:

1. Flock – fly in a flock formation
2. Avoid Obstacles – avoid colliding into Obstacle models
3. Remain In Universe – remain inside the universe (a sphere of user-specified radius)
4. Follow Script – fly along a user-selected path
5. Remain On Course – remain on previous course
6. Fly At Will – fly in a random direction

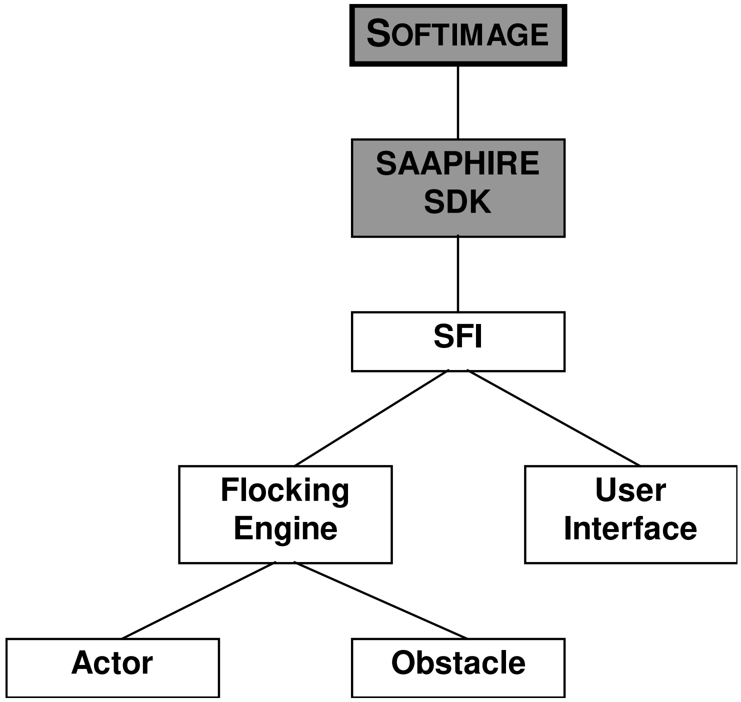
Each instinct produces a direction for the actor to fly in. These directions are then combined to produce a final resultant vector which changes the actor's position in the scene for the next frame. This loop continues until all of the actor's have plotted their courses for each frame, thus producing a flocking animation.

4.1 Initial Generic Implementation

The initial development of the project was to code the flocking system without regards to a specific platform or animation system. The initial system included five major components:

- SFI ("Softimage Flocking Interface") – provides an interface between the generic flocking system and a specific implementation environment
- Flocking Engine – controls the overall generation of the animation; Contains the main loop of the project which asks each Actor to produce results according to its instincts and combines them into a resultant direction of travel for each frame
- User Interface – encapsulates all data and options specified by the user for access by the other components
- Actor – represents a single, independent member of the flock; contains data such as velocity and position, and programmed instincts to control the actor's behaviour
- Obstacle – represents an obstacle (either stationary or moving) which the actors must strive to avoid colliding with

The components were originally arranged as follows:



The SFI component acts as a mediator between the generic flocking code and the code to output the project's results to a specific external system. Therefore, the SFI component consisted of empty function shells. No code was written for them because this initial implementation was for a generic system only. To execute the system and see the results, an external display environment would need to be constructed and the functions of the SFI would have to be "filled in".

This initial implementation forced the author to bang out some frustrating mathematical problems- especially for the Avoid Obstacles and Remain In Universe instincts. It was later discovered that the answers to these problems were somewhat simpler than originally thought. Indeed, some methods were not fully coded until the mathematical puzzle(s) behind them were solved much later.

During this initial implementation, an oversight (one of many) was discovered in the project's design. The project's Design Specifications indicated that a formal object-oriented design was not necessary since the "goodies" that object-orientation provides (e.g. inheritance, polymorphism, etc.) were not required. However, while creating the Actor and Obstacle classes, it became clear that to keep them separate would mean having duplicate data in each class. Both of these classes represent models in the animation scene which share common attributes such as position, velocity and a reference back to that model which each object represents in the scene. Therefore, a Model class was created with these common attributes. The Actor and Obstacle classes are then derived from the Model class and methods appropriate to each are added.

At the same time, this allows methods to pass a Model object instead of either a specific Actor or Obstacle object. Here, polymorphism comes into play in that the position of a model in the scene can be updated without having to specifically pass an Actor or Obstacle.

The initial implementation also used the notion of "global" objects, available to each component, to provide access to the single SFI, User Interface and Flocking Engine objects instantiated. The objects needed to be global to facilitate communication between the components. Also, lists of Actor and Obstacle objects had to be maintained globally for the same reason. This turned out to be a very messy and frustrating practice (as taught in first year Computer Science) and had to mended.

Instead, communication was achieved by passing pointers of these objects to the other objects which needed them. For instance, the Flocking Engine class requires options set by the user (such as the initial and last frame, etc.) For this reason, a pointer to the single User Interface object instantiated is passed to the constructor of the Flocking Engine class where it is saved for future use. This eliminated the need for global objects and data structures.

Since the global lists of Actors and Obstacles had been abolished, two new classes had to be created: ActorArray and ObstacleArray. These classes represent (dynamic) arrays of Actor and Obstacle objects for access (mainly) by the Flocking Engine.

4.2 Windows MFC/OpenGL Implementation

At this point, it was necessary to develop an environment in which to execute and test the initial system. It was decided to implement the first external environment for the Windows platform. The graphical output of the program would be achieved via OpenGL. There are many reasons why this intermediate implementation was created instead of moving straight to the Softimage version. The following provides the rationale for the Windows implementation:

- Easier accessibility – the author had access to a Windows PC in his home- he (unfortunately) did not have such immediate access to a Silicon Graphics workstation equipped with Softimage
- Less crowding – At this time, there existed two (and later, only one) SGI workstations capable of running Softimage at the University. There were at least two other students creating projects requiring these machines at the same time. It was decided it would be more difficult to arbitrate time later, than to use a readily available computer
- SAAPHIRE – The author had only limited experience with SAAPHIRE (Softimage's SDK) learnt mostly from example code and a (questionably useful) reference manual. Using SAAPHIRE (especially learning it) was found to be exceedingly difficult and frustrating. To add this frustration on top of the frustration that would no doubt come from creating this initial implementation would probably have produced unhealthy stress
- Windows programming experience – the author, at the time, had very limited experience with programming the Windows platform- especially using the Microsoft Foundation Classes (MFC). It was decided that this would be an extremely valuable asset to learn, now, in University, for future employment and self-improvement reasons
- OpenGL experience – the author had prior experience only with the SGI-specific Graphics Library (GL) and was keen to upgrade his skills to the modern, platform-independent OpenGL

For these reasons, the Windows version of this project was born. The program was developed using Microsoft Visual C++ 5.0 Professional Edition. It uses the Microsoft Foundation Classes (MFC) to show the program window and dialog boxes used to receive user input. The graphical results are drawn using the Win32 implementation of OpenGL.

Technically speaking, the program is a SDI (Single Document Interface) program and uses MFC's "document/view" architecture. The view class handles all input and output while the document stores the data for each animation. Serialization of the document allows animations to be saved and opened quite easily. An MFC toolbar was added to allow the user to easily: rewind to first frame of animation, play the animation, pause the animation/advance a single frame and to fast forward to the last frame.

There are four dialog boxes in the program. The main one allows entry of all options available to the user in order to control the resultant animation. Two others allow the user to specify the initial positions and sizes of the Actor and Obstacle models in the scene. The fourth is a simple "About" dialog box which gives some brief information whenever the user selects "About…" from the "Help" menu.

Once an animation has been created, the user can rotate his/her point-of-view and zoom in and out by holding the left or right mouse button while dragging the mouse. The playback of the animation can be controlled with the toolbar mentioned above or with similar menu commands. The “Anim” menu also provides some options to reset the view and to toggle drawing the coordinate axes.

While testing the program, it was noticed immediately that the Actors appeared to “teleport” from one side of the universe to the other. It was realized that the distance each Actor could travel in a single frame must be limited. Therefore, an option was added to the user interface to allow the user to specify a maximum acceleration for each frame.

Initially, the Actors were drawn as simple points. This made it hard to distinguish the Actors and to determine their courses. Therefore, code was added to draw a line from each Actor’s previous position to its current position. The line is shaded from dark gray to bright white. This makes each individual Actor’s movements more easily discernible.

Another interesting animation was observed when all but the Remain In Universe and Remain On Course instincts were disabled. The Actors were pushed away from the edge of the universe, through the origin and back again. The movement was smooth and reminded the author of the motion caused by gravity. This was evidence that the non-linear scaling of the instincts was a good idea.

4.3 Softimage Implementation

The Windows implementation of the project allowed the author to test the separate components of the project and to quickly and easily experiment with different settings. However, there were certain options and components which were not implemented such as the ability to pick a path for the Actors to follow. These options just could not implemented easily (imagine entering 20 key-points on a spline for the flock to follow). The time had come to implement the Softimage version of the project with SAAPHIRE to test these abilities.

Between finishing the Windows version of the project and starting with SAAPHIRE, it was realized that the User Interface class, originally envisioned, was unnecessary. Originally, the class was supposed to act as an interface between the Flocking Engine and a specific implementation. The idea was that this class would hold the options set by the user and would hand them out as requested by the Flocking Engine and Actor classes.

However, to make the project as general as possible, and to allow porting of the project to another system by merely changing the SFI class, the User Interface should therefore communicate with the SFI. As a result, the SFI had to provide methods to give the user’s data to the User Interface class. The User Interface class would subsequently have exactly similar methods to provide this information to the other classes. The result was that the User Interface class was simply an extra stop for the required data. Why not “cut out the middle man” and have the other classes ask the SFI directly for the needed information?

Therefore, the User Interface class was abolished and all of its required accessor methods were moved to the SFI class. The deletion has simplified the project by reducing the clutter of inter-class communication.

To learn how to use SAAPHIRE, the author had the SAAPHIRE Reference Manual and some example source code. Both provide good information. What was missing was information on the basics of creating a Softimage custom effect. It was difficult to initially determine how the custom effect was implemented and hooked into Softimage. Some of this important information was buried in the badly organized manual and some was missing altogether. Also, most of the example programs showed how to use SAAPHIRE version 1.0 instead of version 1.1. In general, it seemed as though the manual and example programs assumed the reader had thoroughly read and understood the manuals and examples from SAAPHIRE version 1.0. The author happened not to have access to this information and subsequently, the initial stages of porting the project to Softimage were frustrating and time-consuming.

Eventually, problems and misunderstandings were corrected and the development was completed. To note, the author was particularly dismayed to find that certain features and functions were not available in version 1.1 that were available in the latest version (at the time) of SAAPHIRE, version 1.7. Also, version 1.1 contains a few shortcomings (bugs) that are fixed in the later version. The author was able, albeit via some time-consumption, to “make do” with the early version.

Once familiar with the ways of SAAPHIRE, the project progressed quite well. A description of the additions to the Windows version follows:

After getting the program to run as a Softimage custom effect, a test animation was generated using a number of cone primitives. After running the animation, it was realized that even though the cone models moved about, they always pointed in the same direction (up). The Windows version drew the Actor’s as simple points so no features were available to distinguish their orientation. To create a more realistic animation, the models must point in the direction in which they are travelling. Code was added to rotate the Actor models depending on their positions for the current frame and the frame immediately following it. The mathematical puzzle to accomplish this caused many headaches. However, the solution turned out to be rather simple, leaving the author feeling quite foolish.

Once the Actor models were rotating, it was realized that they were travelling in any direction they wished during each frame without regard to the direction they were travelling during the previous frame. The models rotated at very oblique angles and jumped from point to point. Only in a universe where the Actors had no mass and consequently, no inertia, would this be possible. Therefore, another option was added to the user interface to allow the user to specify a maximum turning angle during each frame. For example, the user could set an angle of 45° so each Actor could not rotate more than this angle between two consecutive frames. Even more mind-numbing mathematics had to be derived to implement this feature (see Appendix A of Technical Manual).

It was noticed that the members of the flock were colliding with one another. The Flock instinct, initially, only encouraged the Actors to fly towards their nearest neighbour. In fact, the Actors were flying too close to one another and bumping into each other. Therefore, a test was added to check the distance to the closest flockmate. If the distance is less than a certain threshold, the Actor is aimed away from its flockmate instead of towards it. This change was very successful. However, this threshold was hard-coded into the program meaning the entire project had to be recompiled to experiment with different values. Clearly, this was unacceptable, so another option was added to the user interface to allow the user to control the density of the flock. This new option simply controls the threshold value.

The Windows implementation had not taken the Actors’ initial headings into consideration. The selected Actors were given initial positions but not headings. This meant that every animation always started with all of the Actor models (e.g. cones) pointing in the same direction (up). It was decided to generate random headings for each model to create more realistic animations.

5. Project Retrospective

5.1 Future work

There is much that could be done to improve the project. The improvements can be classified into two categories: Fixes to difficulties with using SAAPHIRE (version 1.1); and improvements to the project's design and implementation, in general.

Although developing this project as a custom tool for Softimage is a good idea and can yield final results far better than any OpenGL implementation, choosing this course undoubtedly restricts the programmer because he is limited to the features that the SDK provides. This problem came up many times during the Softimage implementation. The following lists some of the features that were not implemented in the current version of the project because of SAAPHIRE limitations or bugs:

- Automatic duplication of Actor models – The project was designed to provide the option to users to simply pick a single Actor model which would then be duplicated and positioned randomly inside the universe, a specified number of times. The author was unable to derive an "easy" way of achieving this. Softimage itself provides a Duplicate→Immediate menu command in the Model module to perform this operation. Why doesn't SAAPHIRE provide a function to use this command? Instead, the author would have been forced to create each target model from scratch by retrieving and duplicating the control vertices that make up the source model. If the source model was complex, that is, if it consisted of a hierarchy of primitives, the task would become even more daunting. It was decided not to implement this option in the hopes that newer versions of SAAPHIRE will support easy duplication of models.
- User interface quirks – Several items about the user interface are a little troublesome to the developer who wishes to be conscious of his lessons learned in the field of Human-Computer Interaction (HCI). Certain annoyances such as having to pick three "dummy" models after picking the actual arguments, and having to pick a "dummy" path for the flock to follow- even though the user does not wish to pick such a path, are disappointing. The first problem can be solved simply by using version 1.7 of SAAPHIRE and changing the trueFlock.cus file slightly. The second is a flaw in the design of Softimage effects in the first place (in the author's opinion). Hopefully, newer versions of the SDK will allow developers to create more user-friendly interfaces.
- The work required to move and rotate the Actor models around the scene was somewhat involved and messy. The developer was forced to work directly with the function curves of the models to set their explicit translation and rotation values for each frame. SAAPHIRE version 1.7 allows the developer to create a spline and to select a model to that spline path. This is exactly like the Path→Pick Path command in the Motion module. Upgrading this project to use the newer version of SAAPHIRE may simplify the animating process somewhat.
- Currently, the user is restricted to flocking only models. That is, Softimage elements such as lights, the camera and the camera's interest cannot be flocked. This is because Softimage doesn't allow the user to collect elements of different types into the same hierarchy. Therefore, the user cannot put (e.g.) both models and lights into a hierarchy to select as the tree of Actors. Possibly, with version 1.7 of SAAPHIRE, the paths of each Actor could be created as a spline (see previous point) so that, after the tool completes executing, the user can put a light or the camera on one of these spline paths.

The following lists items that could be attended to in future versions of the project:

- More sophisticated physical and geometric model – Currently, the project uses a very simple and restrictive physical model: It assumes that all Actor models initially point upwards along the positive y-axis. The project does not currently support forces such as gravity or drag. Nor does it calculate the pitch and roll of the Actors very accurately. A more intricate system would definitely provide a basis for more realistic animations.
- More intelligent flocking – The Flock instinct could use some improving. Presently, the motion created by the tool sometimes resembles a swarm of insects rather than a flock of birds. If the Flock instinct looked at the direction of travel of its nearest flockmate and set a course *alongside* it instead of *towards* it, more realistic flocking might be observed.
- More intelligent collision avoidance – Many of the Actors' instincts could be improved- especially the Obstacle Avoidance instinct. Currently, a simple "force-field model" is used to propel the Actors away from the bounding spheres of Obstacle models. This method fails completely for models which do not fit nicely inside of a sphere. For example, the Actors cannot successfully avoid models such as long pillars and flat rectangular surfaces. Also, the Flock instinct could be improved (complicated): Right now, the Actors only avoid colliding with one another by steering away from their nearest neighbour if they come too close. It would be nice to take other factors into account, such as the direction of travel of the other Actor(s) to determine, with more precision, whether a collision is imminent or not.
- Porting the system – The project could be ported to different platforms and/or animation systems. The most obvious would be to port the project to the Windows NT version of Softimage. SAAPHIRE exists for both the IRIX and NT versions so only small changes should be necessary. Another idea that the author has very seriously considered (and still is) is creating a real-time version of the project implemented as a Windows 95/NT screen saver. The screen saver could flock a number of objects in real-time for a specified number of time, randomly change priorities and other options, and begin another animation.
- The ActorArray and ObstacleArray classes should really come from a single, generic class since they both are exactly the same except that one deals with pointers to Actor objects while the other uses pointers to Obstacle objects.

5.2 Conclusions

The success or failure of this project is not easy to objectively state. The animations created by the initial system looked nothing like a flock of birds in nature. However, with experimentation and time, more *interesting* animations have been produced. The results could be further improved by adding more "intelligence" to the actors' instincts. The results of this project's work can be summarized by the following:

- Intentional patterns of motion, envisioned by the animator, can be created with a decentralized model of autonomous actors
- The model can be constructed in such a way as to be easily adaptable to new systems and modular enough to be easily improved as time, patience and effort are invested
- Combining the model with a powerful, external animation system allows the animator to create much more professional-looking animations for potential scientific, educational and commercial applications

6. References

Articles:

- Dean, Stewart aLife Creatures: An Interactive Essay
- Goldberg, Athomas Behavioral Scripting: Method Acting for Synthetic Characters
- Martin, Allen Particle Systems
- Resnick, Michael New Paradigms for Computing, New Paradigms for Thinking . MIT Media Laboratory
- Reynolds, Craig W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model . Proceedings of SIGGRAPH '87 (Computer Graphics Vol. 21 No. 4, July 1987).
- Reynolds, Craig W. Evolution of Corridor Following Behavior in a Noisy World
- Reynolds, Craig W. Not Bumping Into Things
- Sipper, Moshe An Introduction To Artificial Life
- Terzopoulos, Demetri; Tu, Xiaoyuan and Grzeszczuk, Radek Artificial Fishes with Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World
- Various comp.graphics.animation FAQ

Books:

- Auer, John (1991). Linear Algebra (With Applications) . Scarborough, Ontario : *Prentice-Hall Canada*.
- Hill, F. S. Jr. (1990) Computer Graphics . New York : *Macmillan Publishing*.
- Kreyszig, Erwin (1993). Advanced Engineering Mathematics . Toronto, Ontario : *John Wiley & Sons*.
- Miller, Lawrence H. and Quilici, Alexander E. (1993). Joy of C . Toronto, Ontario : *John Wiley & Sons*.
- Resnick, Mitchel (1994). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds . Cambridge, MA : *MIT Press*.
- Rumbaugh, James *et. all* (1991). Object-Oriented Modeling And Design . Englewood Cliffs, NJ : *Prentice-Hall*.

Internet Resources:

- Beutler, Jan Simple Flocking Algorithm – http://intell-lab.engi.cf.ac.uk/jan/flock.htm
- Kline, Christopher C++ Boids – http://www.tc.cornell.edu/~ckline/boid/boids.html
- Mee, John Flocking Birds – http://www.cs.curtin.edu.au/~mee/boids.html
- Reynolds, Craig W. Boids – http://mht.com/cwr/boids.html
- Sterling, Bruce Artificial Life – http://cs.wpi.edu/~matt/courses/cs563/talks/psys.html