IJCSI

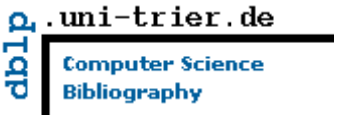
International Journal of Computer Science Issues

A Realistic Rendering of a school of Fish in Openscenegraph and C++

By Séraphin Franclin Foping

Volume 1, 2011
ISSN (Online): 1694-0814

IJCSI proceedings are currently indexed by:



A REALISTIC RENDERING OF A SCHOOL OF FISH IN OPENSCENEGRAPH AND C++

ISSN(online) 1694-0814

A REALISTIC RENDERING OF A SCHOOL OF FISH IN OPENSCENEGRAPH AND C++

by

Séraphin Franclin Foping

MSc Computer Science

University of Yaounde I, Cameroon

Supervisors

Dr Paul Chapman

and

Mr Kim Bale

Department of Computer Science

University of Hull

September 2007

CONTENTS

Abstract **viii**

Acknowledgements **ix**

Dedication **x**

1 Introduction **1**

1.1 Motivations 1

1.2 Initial specification 2

1.3 Aims and objectives of the project 2

1.4 Structure of the dissertation 2

2 Background **4**

2.1 Related work 4

2.1.1 The flocking behaviour 4

2.1.2 Previous implementations 5

2.2 Overview of an underwater scene 6

2.3 Rendering underwater environments 7

3 Designing the flocking algorithm **10**

3.1 Reynolds' flocking algorithm 10

3.1.1 Flocking as a behavioural model 11

3.1.2 Presentation of a boid 11
3.1.3 The separation rule 12
3.1.4 The alignment rule 12
3.1.5 The cohesion rule 14
3.2 Applying constraints on boids 15
3.3 The obstacle avoidance rule 16
3.4 Computing orientation 17
3.5 Algorithmic considerations 18
3.6 Applications 19
3.6.1 In robotics 19
3.6.2 In the music industry 19
3.6.3 In the video and games industries 20

4 Improving the realism of the scene 21
4.1 Fog 21
4.2 Fish modelling 22
4.2.1 Designing the fish body 22
4.2.2 Fish motion 23
4.3 Adding grass on the terrain 23
4.4 Modelling caustics 24

5 Implementation 26
5.1 Software library 26
5.2 The scene graph approach 27
5.2.1 Overview of graphs 27
5.2.2 Definition 27
5.2.3 Benefits 28
5.3 Design patterns 30
5.3.1 The observer pattern 30
5.3.2 The visitor pattern 30
5.4 Description of OpenSceneGraph 31

5.4.1 The core OSG library 32
5.4.2 The processing pipeline 33
5.4.2.1 The update traversal 33
5.4.2.2 The cull traversal 33
5.4.2.3 Render traversal 34
5.5 Dynamic modification of a scene gaph 34
5.6 The coding of the flocking algorithm 35
5.6.1 OSG classes 35
5.6.2 Flocking implementation classes 36
5.6.3 Utility classes 37
5.6.4 Flocking algorithm revisited 38
5.7 Optimizing the scene graph 38
5.8 Major issues 39
5.8.1 OSG issues 39
5.8.2 Flocking algorithm drawbacks 39

6 Software evaluation 40
6.1 Evaluation of the software 40
6.1.1 Results 40
6.1.1.1 Fish shoaling 40
6.1.1.2 Obstacle avoidance 41
6.1.1.3 Terrain visualization 41
6.1.1.4 Caustics 41
6.1.1.5 Underwater visualization 42
6.2 Performance of the program 42
6.3 Comparison with other implementations 45

7 Critical appraisal 50
7.1 Objectives summary 50
7.2 Project management 51
7.3 Lessons learnt 51

8 Conclusion and future works **53**

Appendices **54**

A The flocking implementation class diagram **55**

B The scene graph sketch **56**

C User manual **57**

C.1 Requirements 57

C.1.1 Software 57

C.1.2 Hardware 57

C.2 Installing the software 58

C.2.1 Controls 58

C.2.1.1 Mouse 58

C.2.1.2 Keyboard 58

C.3 Compiling the source code 59

C.4 Recommended configuration 60

LIST OF FIGURES

2.1 Boids 9

3.1 A boid 11
3.2 The separation rule 12
3.3 The alignment rule 14
3.4 The cohesion rule 15
3.5 Euler angles 17

4.1 A fish mesh 22
4.2 Fish body motion 23
4.3 A grass on the terrain 24

5.1 A scene graph 28
5.2 OSG Functional Components 32
5.3 OSG pipeline processing 33
5.4 A scene graph containing a flock of three fish 36

6.1 A school of fish 41
6.2 The obstacle avoidance rule 42
6.3 The terrain 43
6.4 Caustics effect 43
6.5 The caustics map 43

6.6 The caustics effect from Finding Nemo 44
6.7 The final rendering 47
6.8 Statistics of the simulation 48
6.9 A school of fish from PSCrowd 49

A.1 The simplified class diagram 55

B.1 The scene graph sketch 56

LIST OF ALGORITHMS

1 Flocking algorithm 10
2 The separation rule 13
3 The alignment rule 13
4 The cohesion rule 15
5 Limit Velocity 16
6 Obstacle avoidance 16

Abstract

Reynolds first simulated the flocking behaviour in a program called Boids. His algorithm consists of three simple rules: separation under which every boid should steer to avoid crowding neighbours, alignment states that every boid should match the velocity of its neighbours and the cohesion emphasizes on the fact that every boid should head themselves towards the centre of the flock. This report focuses on implementing that flocking algorithm on three-dimensional fish models in OpenSceneGraph. The simulation will make use of callbacks. The result will be a real-time simulation of a school of fish. Users will be able to dynamically switch on and off the flocking behaviour. In order to increase the realism of the environment, items such as grass, caustics, fog, bubbles and terrain will also be considered.

Acknowledgements

- First of all, I wish to express my gratitude to the **University of Hull** and the **Department for International Development (DfID)** for offering me this wonderful opportunity to study in the United Kingdom.
- I also wish to thank all the Computer Science Department staff for their availability and their advice which were of great help. May I thank especially my project supervisors **Dr Paul Chapman** and **Mr Kim Bale** for their availability and their patience, my personal supervisor **Dr Helen Wright** for her attention, concern, advice and availability and finally my lecturers **Mr Warren Viant, Mr Derek Wills, Dr Jon Purdy** and **Mr Darren McKie**.
- I would like to thank my parents **Mr Séraphin Foping** and **Mrs Christine Foping** back in Cameroon for their support.
- I would like to thank my brothers, sisters and friends in Cameroon.
- Finally, I would like to express my gratitude to all my classmates for their permanent support throughout the academic year. Cheers, guys!

Dedication

To the Almighty God

CHAPTER

ONE

Introduction

One of the most challenging tasks in computer graphics is to be able to render realistic underwater scenes. In the following lines, answers to the following questions will be addressed:

- Why do we need to simulate underwater environments?
- What are the main properties of such environments?

1.1 Motivations

Over the past few years, there has been an increased interest in deep-sea underwater exploration and engineering. (Chapman, Conte, Drap, Gambogi, Gauch, Hanke, Longand, Loureiro, Papini, Pascoal, Richards & Roussel 2006) described a project aiming at the virtual exploration of underwater sites (**VENUS**). Such exploration enables everyone to learn in depth about archaeological sites in a safe and cost-effective environment. Hence the needs to model a physically accurate simulation.
Realistic underwater rendering could be used to allow divers and submersible remotely operated vehicle pilots to predict the visibility and analyse optimum

equipment and procedures for a range of underwater conditions. Moreover, recent attempts at underwater scenes within the entertainment market, although believable, are of limited physical accuracy and are often time consuming to create.

1.2 Initial specification

The initial specification of the project states that it will focus on rendering realistic underwater environments for use in an underwater archeology project, the VENUS project described in section 1.1. Items to be considered will include silt rendering, fog and fish schooling.

1.3 Aims and objectives of the project

As it is mentioned in the initial specification, this project should focus on rendering underwater scenes. However as the specification were tightened up, my supervisor advised to focus only on the flocking behaviour of fish.
Therefore, the project aims at applying Reynolds' flocking algorithm in OpenSceneGraph (OSG) on three-dimensional fish models in order to simulate a school of fish. A realistic underwater rendering should be provided. Items such as caustics, grass, underwater terrain and fish body motion should be addressed.

1.4 Structure of the dissertation

The dissertation has been broken into nine parts:

- The first part of the report (chapter 2) will be a presentation of the background.

- Reynolds' flocking algorithm will be unwound in chapter 3. The chapter will be concluded by some applications of the flocking behaviour.
- The design part of the report will be closed by presenting algorithms used to increase the realism of the scene will be explained in chapter 4.
- Having described in depth the design process of both the flocking and the underwater scene, the implementation part of the algorithm will be presented in chapter 5. Before describing OSG in depth in section 5.4, scene graphs will be addressed in section 5.2.
- The software will be evaluated in chapter 6.1. Results and comparisons with previous implementations will also be detailed in this chapter.
- A critical appraisal and the project management will be discussed in chapter 7.
- The report will be concluded in chapter 8 and future research areas will be given.
- A simplified version of the class diagram of the software will be presented in appendix A.
- Finally, the scene graph of the software will be presented in appendix B and the user manual will be detailed in appendix C. The minimum configuration to run the software will also be discussed in that appendix.

CHAPTER

TWO

Background

2.1 Related work

2.1.1 The flocking behaviour

Reynolds (1987) described an agent-based simulation of flocking in which each member of a flock (called *boid*) can be defined in terms of interacting particle systems. He showed that the complexity of his algorithm is $O(n^2)$, and suggested spatial hashing as a means of improvement. His first implementation was an off-line process.

Tu & Terzopoulos (1994) describe a much more advanced algorithm for their artificial marine life. Carlson & Hodgins (1997) present a method of reducing the computational cost of simulating groups of animals by using less accurate simulations for individuals when they are not important to the viewer. They also present a system to decrease the computational cost of the motion of a herd of one-legged creatures. Gabbai (2005) suggested that flocking has been considered as a means to control the behaviour of unnamed air vehicles.

Okubo (1986) suggested that the coordination in flocks might be achieved by the application of the mathematics of nonlinear dynamics. Heppner (1987)

suggested that the flocking behaviour may be an emergent property arising from individuals following simple rules of motion. Later on Heppner & Grenander (1990) presented a computer flight flock simulation where the flock was a self-organizing aggregation of individuals, whose behaviour was based on stochastic nonlinear differential equations. Both simulations assumed common grounds and model the behaviour on the clues of attraction and repulsion.

2.1.2 Previous implementations

There were several attempts to simulate the flocking algorithm. Boids, the first computer program to simulate the flocking behaviour, was written by Reynolds (1987) in Symbolics Common Lisp, and was based on Symbolics' S-Geometry 3D modeling system and S-Dynamics animation system. Reynolds (2006) also presented an implementation of his flocking algorithm for the PlayStation®3 called PSCrowd.

Courty & Musse (2005) also implemented the FastCrowd system which were able to run a crowd of 5,000 individuals at about 100 frames per second, and a crowd of 10,000 at 35 frames per second (without visualization, 50 frames per second and 20 frames per second with individuals drawn as two-dimensional disks). The graphics processing unit (GPU) also computed the flow of smoke for fire evacuation scenarios. The FastCrowd made use of general purpose computation on GPU algorithms and the flock simulation is computed by using both the central processing unit (CPU) and the GPU.

Erra, Chiara, Scarano & Tatafiore (2004) described a GPU implementation of Boids which is entirely computed on an nVIDIA® NV35 graphics card, thereby exploiting the intrinsically parallel nature of the flocking behaviour. Their system also made use of a *scattering matrix* to detect when the flock departs from mainly parallel flight.

Chapman, Viant & Munoko (2004) also implemented Reynolds' flocking algorithm in OSG.

Woodcock (2000) described an implementation of Reynolds' flocking algorithm in C++. His system did not take into account the obstacle avoidance rule, which was added later by Reynolds. Boids are stuck in the flocks they started with. Woodcock (2001) released a more improved version of his program with a predator and prey rule. Performance and analysis of all these approaches will be presented in section 6.1

2.2 Overview of an underwater scene

A realistic underwater scene requires to taking into account many other aspects such as the wildlife, optical effects like caustics and shafts of lights. In order to increase realism of the scene, one has to focus on fish and their behaviours such as their ability to move, sense and think.
Accurate simulations of fish behaviours have hinted at the uses of artificial life algorithms. The most interesting behaviour falling in that category is certainly the shoaling motion of an aggregation of fish. This particular behaviour as well as other have been extensively studied in artificial life. The flocking algorithm first created by Reynolds (1987) is the most obvious one to base fish schooling behaviour on. He introduced a distributed agent based flocking model in which each flock member or boid follows three basic rules. He described this model is an extension of the particle systems introduced by Reeves (1983). Figure 2.1 shows a screenshot of Boids.

2.3 Rendering underwater environments

The main issue with such environment is the density of water which is up to eight hundred times higher than air. That density drastically increases with depth. With increasing depth, sunlight is absorbed hence reducing the visibility. This brings about a new way of light interactions and the need of special rendering approach. Aranha (2005) suggested that the interaction with water molecules and suspended particles caused effects, including loss of contrast, diffusion of rays, change in colour and reduction of intensity. Light scattering causing optical effects such as caustics and shafts of light due to atmospheric particles is also an important part for any underwater rendering.

Another issue is the exponential increase of the pressure in respect to the depth. For instance at about ten meters below the surface, the water exerts twice the pressure on a human body as air at surface level. For heavy objects, this is not an issue because they will be provided with the buoyancy which tends to make them feel lighter.

Pixar (2003) produced a computer-animated film with one of the most impressive demonstration of underwater effects in the movie Finding Nemo (Pixar (2003)), which was created off-line. To create these graphics, Pixar's Renderman was adapted to blur objects based on ocean depth and distance from a specific viewpoint. Water interaction was created by modifying Fitz, Pixar's fur and cloth simulator (Deoswitz 2003). Several algorithms for rendering caustics, shafts of light and the colour of the water have been published for underwater images. Shinya, Saito & Takahashi (1989) proposed an algorithm for rendering caustics. The rationale is to compute the illumination distribution on the surface in advance by using grid-pencil tracing. Briere & Poulin (2001) displayed caustics by using a hierarchical light beam structure. The main drawback of all these algorithms is their reliabilty on ray-tracing, making them slow. Finally Iwasaki, Dobashi & Nishita (2002) proposed a fast method for rendering underwater optical effects using advanced graphics

hardware. The rationale is to use illumination volumes for displaying shafts of light. Deusen, Ebert, Fedkiw, Musgrave, Prusinkiewicz, Roble, Stam & Tessendorf (2004) presented an algorithm to render sea animation which was not based on physics models but instead used statistical models based on observations of the real sea. Their method had been extensively used for commercial purposes, mainly for sea animation in the movies Titanic and Waterworld.

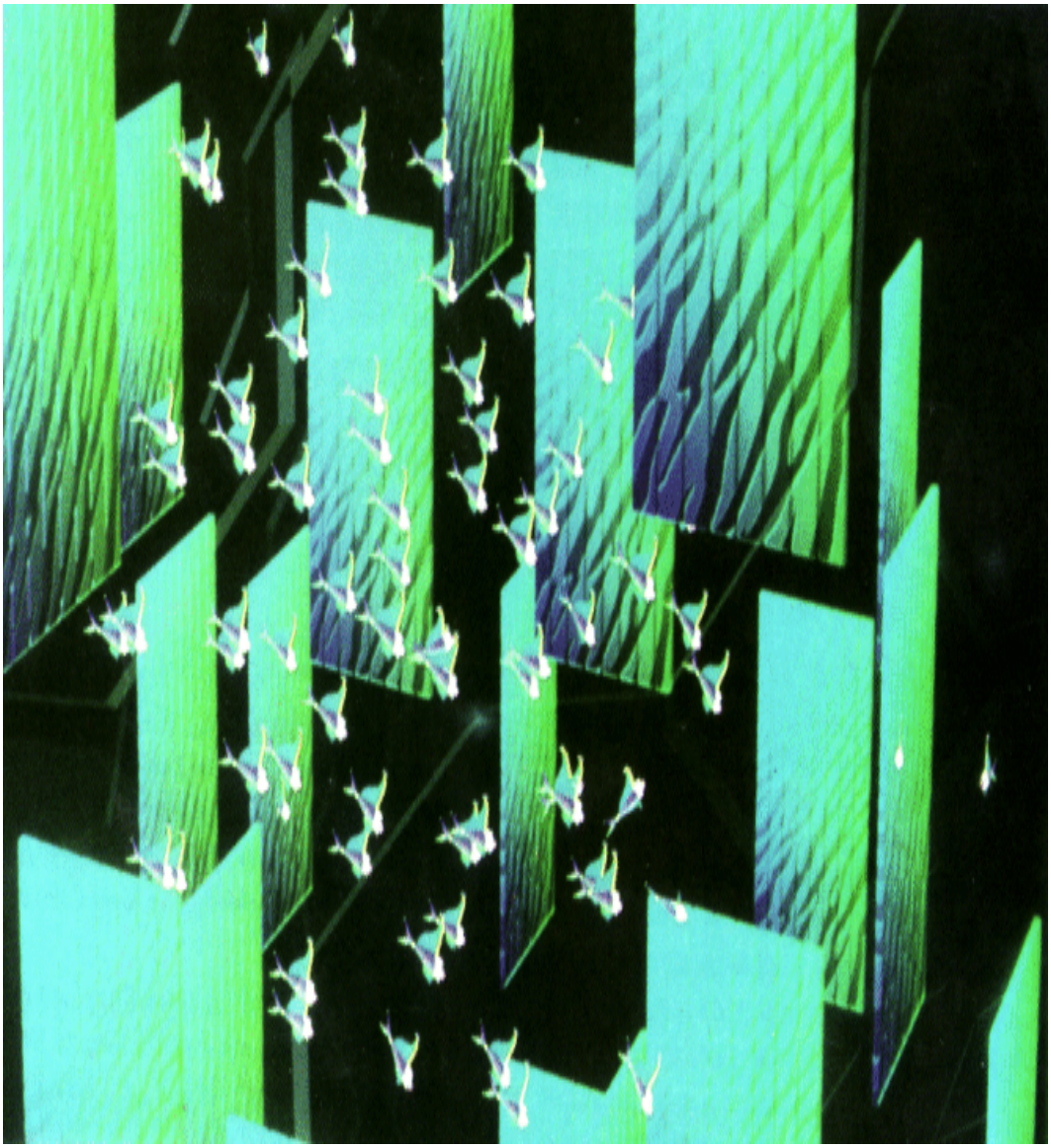


Figure 2.1: Boids (Reynolds (1987))

CHAPTER

THREE

Designing the flocking algorithm

3.1 Reynolds' flocking algorithm

Flocking is a good example of emergent collective behaviour. The main characteristic of this model is the lack of a group leader. Flocking behaviour emerges from local interactions. Each agent, a member of the flock, has direct access to the geometric description of the world, but reacts only to its nearby flock-mates. The basic flocking model consists of three simple rules: cohesion, alignment and separation.
Listing 1 shows a pseudo-code of Reynolds' flocking algorithm.

Algorithm 1 Flocking algorithm

```
PROCEDURE FLOCKING
initialise_boids()
LOOP
  apply_rules_on_boids()
  apply_constraints()
  calculate_rotation_angles()
  draw_boids()
ENDLOOP
END PROCEDURE
```

3.1.1 Flocking as a behavioural model

Reynolds used the word *boid*, a contraction of birdoid, to refer to a member of a flock. It should be noted that each boid has only a local knowledge of the world. This knowledge comes from a simulated vision from its current position. Hence the lack of a leader in a flock. The entire flock decides in a distributed manner in order to get a synchronized and smooth motion. Reynolds observed that, none of the creatures of a flock has a full knowledge of the entire group, meaning that every member of the flock can only perceive its nearby flockmates. The aggregation of all these behaviours bring about the complex motion as seen in nature. The bulk of the simulation of this model is the distributed partial knowledge of the group. In order to achieve a realistic movement, rotation equations should be computed in order to allow the boid to rotate itself with respect to its velocity.

3.1.2 Presentation of a boid

A boid can be defined as a set of parameters used to simulate the flight as mass, maximum speed, maximum acceleration, global position, current speed and a view reference system used to represent its the point of view. Figure 3.1 shows a boid and its coordinate system.

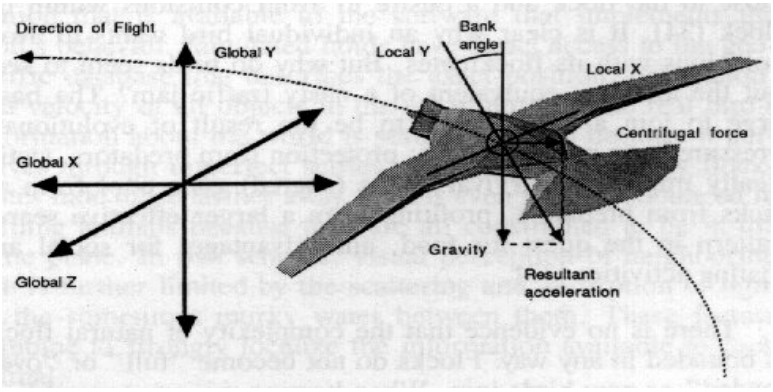


Figure 3.1: A boid (Reynolds (1987))

Part of boids' information is constant and defined with initial conditions while others need to be updated every frame hence the discrete nature of the simulation.

3.1.3 The separation rule

This rule allows a boid to maintain a distance (threshold) from its nearest flock-mates. When a boid is within a certain distance from one of its neighbours, it is repelled from it so as not to collide with them. As the boid moves closer to the nearby agent, the force of repulsion increases proportionally to the square of the distance between the agent and its vicinity. Listing 2 details the pseudo-code of this rule and figure 3.2 illustrates the rule.

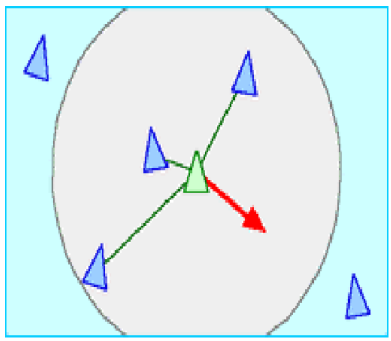


Figure 3.2: The separation rule (Reynolds (1987))

3.1.4 The alignment rule

This rule enables the flock members to go in the same direction and at the same speed as their nearby flock mates to travel in the same direction and speed. Nearby flock mates cause the boid to steer more to match their velocity. The relationship is proportional to the square of the distance between them. Figure 3.3 and listing 3 illustrates and details the pseudo-code of this rule respectively.

Algorithm 2 The separation rule

INPUT: *Boid* a_i
OUTPUT: New Position
  ALGO SEPARATION(BOID a_i)
  $Vector3D \;\; NewPosition = 0$
  FORALL BOID a **DO**
    IF $(a \neq a_i)$ **THEN**
      IF $|a_i - a.position| \leq \; THRESHOLD$ **THEN**
        $NewPosition \leftarrow NewPosition - \; (a_i.position \; - \; a.position)$
      ENDIF
    ENDIF
  ENDFOR
  RETURN NewPosition
  END ALGO

Algorithm 3 The alignment rule

INPUT: *Boid* a_i
OUTPUT: New Velocity
  ALGO VMR(BOID a_i)
  $Vector3D \;\; PVelocity_i$
  FORALL BOID a **DO**
    IF $(a \neq a_i)$ **THEN**
      $PVelocity_i \leftarrow PVelocity_i + a.velocity$
    ENDIF
  ENDFOR
  $PVelocity_i \leftarrow PVelocity_i/N - 1$
  RETURN $(PVelocity_i - a_i.velocity)/SCALEFACTOR$
  END ALGO

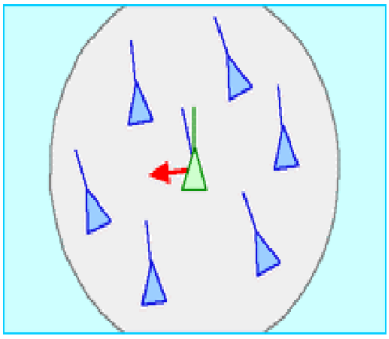


Figure 3.3: The alignment rule (Reynolds (1987))

3.1.5 The cohesion rule

This behaviour makes a boid to be near the centre of the flock. It should be noted that each boid's notion of the centre of the flock is a local centre. It is actually the centre of the nearby flock mates.
The flock centring urge depends on where the boid is in relation to the rest of the flock. At the centre of the flock, its neighbours being approximately evenly distributed about the boid, the flock centring urge is low. At the outside of the flock, the boid's local flock mates are more distributed towards the inside of the flock and the flock centring behaviour causes the boid to steer towards this centre. The farther away the boid is from the flock centre the more it is attracted to it. The force of this attraction is proportional to the square of the distance. Figure 3.4 and listing 4 illustrates and details the pseudo-code of this rule respectively.

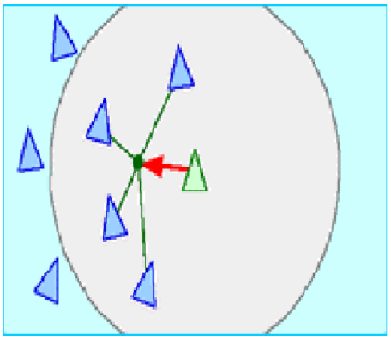


Figure 3.4: cohesion rule (Reynolds (1987))

Algorithm 4 The cohesion rule

INPUT: *Boid* a_i
OUTPUT: New Position
ALGO FC(BOID a_i)
$Vector3D \;\; Position_i$
FORALL BOID a **DO**
IF $(a \neq a_i)$ **THEN**
$Position_i \leftarrow Position_i + a.position$
ENDIF
ENDFOR
$Position_i \leftarrow Position_i/N - 1$
RETURN $(Position_i - a_i.position)/SCALEFACTOR$
END ALGO

3.2 Applying constraints on boids

Having computed the heading of boids, their velocity and acceleration vectors have to be constrained in order to prevent them from an un-realistic motion. This is achieved in listing 5.

Algorithm 5 Limit Velocity

INPUT: *Boid* a_i
OUTPUT: New Velocity
ALGO LimitVelocity(BOID a_i)
$Vector3D\ \ Velocity$
$Normalize(Velocity)$
$NewVelocity = Velocity * SCALEFACTOR$
RETURN $NewVelocity$
END ALGO

3.3 The obstacle avoidance rule

This rule was added by Reynolds (2000). Its aims at avoiding any obstacles in the world. In this project, the obstacles were the boundaries of the bounding volume. This will ensure that boids always travel within the specified bounding box representing the world. The corresponding algorithm is as follows:

Algorithm 6 Obstacle avoidance

INPUT: *Boid* a_i $AND\ BOUNDING\ BOX$
OUTPUT: New Position
ALGO LimitPosition(BOID a_i)
$Vector3D\ \ Position$
IF (Position IS CONTAIN in the World) **THEN**
$NewPosition = 0$
ENDIF
IF (Position IS OUTSIDE the World) **THEN**
$NewPosition = Position - AMOUNT$
ENDIF
RETURN $NewPosition$
END ALGO

3.4 Computing orientation

Another important feature is the rotation of boids. Ideally, they should be able to compute their rotation angles based on their velocities. The rotation of a particular boid was defined by the means of Euler angles and quaternions. Lander (1998) described Euler angles and suggested the use of quaternions to avoid issues such as gimbal lock, interpolations and round-off errors (Dunn & Parberry (2002)).

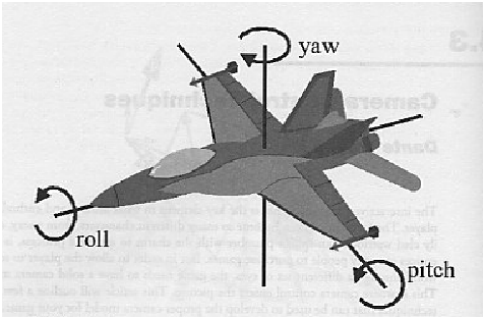


Figure 3.5: Euler angles (Woodcock (2000))

The rotation of boids were defined by only two angles: the heading and the pitch. The heading defines the rotation around the up vector (the z-axis in OSG) whereas the pitch defines the rotation around the y-axis[1]. By using simple flight dynamic equations, the following equations were used to compute the heading and the pitch angles:

$$\boldsymbol{Pitch} = -\arctan(\frac{\boldsymbol{Velocity.Z}}{\boldsymbol{Velocity.X}}) \tag{3.1}$$

$$\boldsymbol{Heading} = -\arctan 2(\boldsymbol{Velocity.X}, \boldsymbol{Velocity.Y}) - \frac{\boldsymbol{\pi}}{\boldsymbol{2}} \tag{3.2}$$

[1]OSG uses a left-handed coordinate system with the up-vector being the z-axis, the positive y-axis pointing into the screen while the x-axis points at the right of the user in the horizontal direction.

The value of the pitch returned by the equation 3.1 will always belong to the range $\left[-\frac{\pi}{2}\ ,\ \frac{\pi}{2}\right]$ which is the definition set of the **arctan** function. This means that there is no need to constraint the angle. After all, in order to use canonical Euler angles, the pitch must be defined from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ excluded. Furthermore, the value of the heading returned by the equation 3.2 will also fall to the range $[-\pi\ ,\ \pi]$ which is exactly the required value for the heading. All these will make sure canonical Euler angles are used.
Having worked out those angles, they were converted into quaternions before being applied to the rotation functions. Quaternions present the following advantages:

1. Smooth interpolation
2. Fast addition
3. Fast conversion to matrix form
4. Less memory usage

However, Dunn & Parberry (2002) highlighted some drawbacks of quaternions.

3.5 Algorithmic considerations

The three basic rules allow boids to exhibit a flock-like motion. In the flocking system, the algorithm checks each boid against every other agent in the world to either include it as a nearby neighbour or disregard it. Therefore the algorithm has an asymptotic complexity of $\boldsymbol{O(n^2)}$ where $\boldsymbol{n}$ is the number of boids in the world. Reynolds later presents an approach to improve the neighbouring query in the flocking algorithm.
However it is also possible to improve that complexity to nearly $\boldsymbol{O(n)}$ by using a suitable spatial data structure which allows each boid to be kept sorted by their location. Therefore, finding the nearby flockmates of a given boid

requires examining only the portion of the flock which is within the general vicinity. By using such algorithmic speed-ups and modern fast hardware, large flocks can be simulated in real time, paving the way for interactive applications.

3.6 Applications

Flocking algorithms have extensively been used in robotics, musical and computer animation industries.

3.6.1 In robotics

Kelly (1995) described a process to build a robot that can act as an insect. He called it Allen. The oustanding feature of Allen was its ability to learn, in an evolutionary time, to move through a complex world by building up a set of behaviours such as: avoid collision with nearby objects, wander aimlessly, navigate the world, compute an internal map, being aware of any environmental alterations, express any travel plans and dynamically change plans.
The artist Leonel Moura also used a flocking algorithm to make a robot simulation, he also showed that a spider web, a honeycomb or a shell pattern reveals complex designs. He concluded that all these happened as a result of emergence and self-organization and ended up with a translation of flocking based algorithms.

3.6.2 In the music industry

In order to award the *Original Musical Score*, the panel uses a software that treats music as a type of 3D space, in which the dimensions are pitch, loudness and note duration. As musicians perform, a swarm of digital particles

immediately starts to buzz around the notes being played in this space - in the same way that bees behave when they are seeking out pollen.
Swarm Music described by Blackwell (2005) is an improvisation and composition system inspired by the behaviour of insect swarms. The individuals in Swarm Music, however, are not bees or ants, but musical events. A musical event inhabits Music Parameter Space (MPS) where each dimension corresponds to a musical variable such as pitch, pulse, duration and loudness. Events are attracted towards each other and to events left behind by other swarms. And like real swarms, individuals also take care not to bump into each other.
As the events swarm around each other, constantly varying melodies, harmonies and rhythms are produced. Attracting events captured from an external source enable humans to interact with the swarm. Insect swarms have a remarkable abilility to organise themselves despite being leaderless and the simplicity of each individual. This phenemenon is called self-organisation, and Swarm Music demonstrates that these principles apply to music too.

3.6.3 In the video and games industries

Tim Burton's *Batman Returns* released in nineteen ninety-two contained a computer simulated bat swarms and penguin flocks which were created with modified versions of the original boids software developed by Reynolds (1987). Disney's *The Lion King* released in nineteen ninety-four also included a wildebeest stampede.
In the games industry, the flocking behaviour provides an interesting tool for aggregate motion. Many commercial titles made use of it. For instance, Epic's *Unreal* and Sierra's *Half-Life* used flocking algorithms for many of their monsters as well as other creatures such as fish and birds. *Enemy Nations* of Windward Studios used a modified flocking algorithm to control unit formations and motion accross the space.

CHAPTER

FOUR

Improving the realism of the scene

4.1 Fog

Fog can be defined as an atmospheric effect, its presence in a scene greatly increases its realism. Fog is also a depth cue since it depends on the distance from the camera, therefore hinting how far away objects are. When used properly, it can be a key asset for the culling process by providing a smoother culling of objects by the far plane. This is why OSG manages it within the cull traversal, recall from the chapter 5 that all state managements of a scene are handled during the cull traversal.

There are two ways to implement the fog in OSG: linearly or exponentially. The former method can be achieved by blending the color of the pixel with the colour of the fog of the scene. The blending factor is a function of the distance from the viewer to the camera. In addition to that, there is a fog factor to decrease linearly with the depth from the viewer by using the following parameters: the starting distance of the fog and the end distance of the fog.

In order to achieve all this in OSG, the state management of the root node was altered so that every children of this node will automatically inherit from

this attribute. Results will be presented in chapter 6.1.

4.2 Fish modelling

4.2.1 Designing the fish body

The body of a fish is made up of a combination of a skeleton with a texture mapped fish mesh. The actual process of mapping a skeleton onto a texture mapped fish mesh requires the import of an appropriate fish mesh into a 3D modelling tool such as **Autodesk^{TM} 3D Studio Max^{TM}**, mesh texture mapping, and the binding of the skeleton to the mesh.
The skeletal model of fish used within this project is an improvement of the one used by Gates (2002). Figure 4.1 shows a screenshot of its wireframe mode as well as the number of primitives and vertices contained by the fish model (a perch). The fish model is made up of ten drawable objects or sub-meshes: two for its body (front and back faces), two for its eyes, two for its vertical and dorsal fins, one for its tail and another one for its head. Throughout this project all 3D fish models were obtained from Toucan (2007).

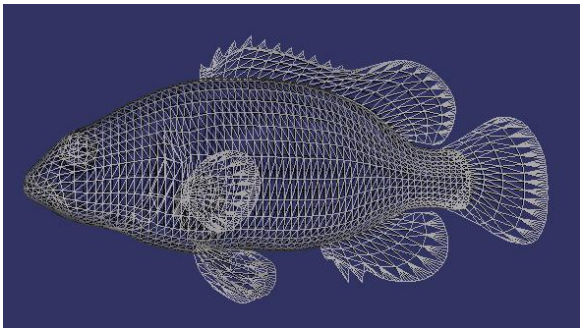


Figure 4.1: A fish mesh

4.2.2 Fish motion

Gates (2002) described an algorithm to simulate the swimming motion of fish. Quan, Zhong, Wang, Xu, Liang & Zhang (2006) also described an algorithm to model the fish body motion. They applied a distortion function to its body, therefore the fish model is constructed by a "particle-spring-damp". In order to implement it in OSG, a vertex shader was used. The rationale is to apply a travelling sine wave function to its vertices. The corresponding pseudo-code is as follows:

1. Retrieve the position of vertices.
2. Displace the $\boldsymbol{y}-$component by applying a travelling sine wave function to it.
3. Render the fish.

Figure 4.2 shows the result.

Figure 4.2: Fish body motion

4.3 Adding grass on the terrain

The addition of grass to the scene greatly increases its realism. A simple ray-tracing algorithm was used in order to work out the correct height on

the terrain. Its corresponding pseudo-code is as follows:

1. From the viewer to the desired position on the terrain, a ray is fired off.

2. The height is computed by considering the magnitude of the vector whose edges are the desired position and the intersection of the ray with the terrain

This process was implemented in OSG using intersections. Figure 4.3 shows the result.

Figure 4.3: A grass on the terrain

4.4 Modelling caustics

Applying caustics to the scene is another key asset to increase the realism of the scene. Several works have been done in the field. Wand & Strasser (2003) described an algorithm to render caustics in real-time. However, achieving a real-time simulation of underwater lighting is still one of the most challenging tasks in computer graphics. This is mainly due to the density of the water which is around eight hundred times greater than the air.
Caustics occur when light is reflected at the seabed, focused into ray bundles

of a certain structure, and then received as patterns of light on a diffuse surface. Caustics are a subtle effect but for underwater scenes, these small details are crucial to obtain realistic images.
Stam (1996) also presented an algorithm to simulate the caustics effect. The rationale was to model caustics through textures synthesized using a wave description of the propagation of the light. His algorithm can be summarized as follows.

1. Generate random phase from a statistical description of the surface.
2. Transform resulting wave into Fourier domain.
3. Perform multiplication with the filter.
4. Inverse Fourier transform the result.

Throughout the project an attempt to simulate this effect was achieved in OSG. It was done on the GPU using a multitexturing technique. A noise texture was defined as the second texture unit. The caustics effect consisted of applying a slow and smooth motion of that noise texture by altering the texture coordinate in one dimension, namely the horizontal direction.
Results of these will be presented in chapter 6.1.

CHAPTER

FIVE

Implementation

This chapter will focus on the actual implementation process of the flocking algorithm in OSG. The full description of the required software library will be given in section 5.1, further details about scene graphs will be presented in section 5.2.
Furthermore, description of some behavioural patterns, namely the visitor pattern and the observer pattern will be presented in section 5.3. The chapter will be closed by an overview of OSG in section 5.4 and also **callbacks** which have been used to update every node of the scene graph.

5.1 Software library

The successful completion of the code would have not been possible without using the following programs:

- **Autodesk^{TM} 3Ds Max^{TM} 9** was used for modelling objects such as fish and terrain.
- **ATI^{TM} RenderMonkey^{TM} 1.6** provided the shader framework. It was used to model the fish body motion, caustics and floating plants.

The actual process to achieve it will be described in chapter 4.

- **Microsoft^{TM} Visual Studio^{TM} 2005** provided the core framework to code the flocking algorithm in C++.
- **OpenSceneGraph 1.2** provided the necessary graphics toolkit to achieve the program. Because of its crucial role to the project, its description will be given in section 5.4.

5.2 The scene graph approach

5.2.1 Overview of graphs

A graph is a data structure, an abstract data type consisting of a set of nodes and a set of edges describing the connections between nodes. Foping (2006) studied in depth planar graphs and also described an optimal algorithm to traverse them. Scene graphs consist of a graph of nodes depicting the spatial sketch of a 3D scene while hiding graphic characteristics in objects. Hence the strengths of scene graphs; spatial organization for culling and embedding the entire scene in a scene graph. In addition to that, there is a need to have a powerful, easy-to-use and scalable application. The next lines will focus on the definition of scene graphs as well as giving some examples.

5.2.2 Definition

A scene graph is a data structure aiming at arranging the logical and often spatial representation of a graphical scene. It is typically drawn with the root at the top, and leaves at the bottom. It starts with a top-most root node which encompasses the whole virtual world. The world is then broken down into a hierarchy of nodes representing either spatial groups of objects, settings of the position of objects, animations of objects, or definitions of logical relationships between objects such as those to manage the various

states of a traffic light. The leaves of the graph represent the physical objects themselves, their drawable geometry and their material properties. Formally speaking, a scene graph can be defined as a tree or as a directed acyclic graph with a random number of children. The following diagram shows an example of a scene graph representing the solar system.

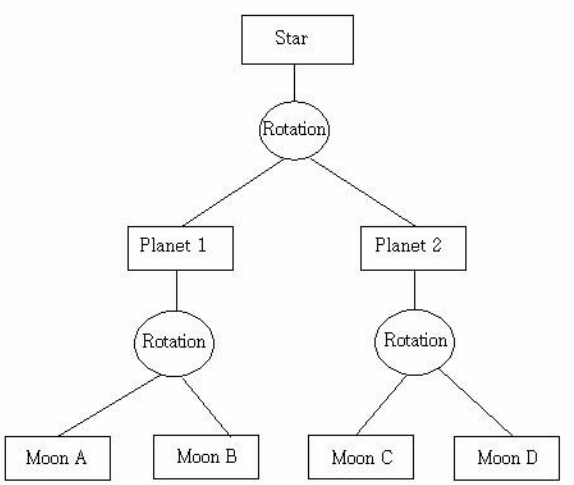


Figure 5.1: A scene graph

5.2.3 Benefits

- Performance: scene graphs provide an excellent framework for optimizing graphics performance. A good graphics engine should encompass the culling of objects that will not be sent to the graphics pipeline for rendering on screen, and state sorting of properties such as textures and materials, so that all objects that are alike are rendered together. By not supplying this technique, both the CPU and the GPU will be swamped by a tremendous amount of data. Scene graphs make this process less painful by supplying a mechanism to sort the state of objects.

- Productivity: scene graphs take away much of the hard work required to develop high performance graphics programs. The engine manages all the low-level graphics, reducing what would be thousands of lines of OpenGL down to a small number of simple calls. Furthermore, one of the most powerful concepts in object-oriented programming is object aggregation, enshrined in the composite design pattern, which perfectly fits in the scene graph tree structure, hence making a highly flexible and reusable design. Scene graphs also often come with additional utility libraries which range from helping users set up and manage graphics windows to importing 3D models and images. A dozen lines of code can be enough to load mesh data and create a real-time simulation.

- Portability: scene graphs encapsulate much of the lower level tasks of rendering graphics and reading and writing data, reducing or even removing the platform specific codes used in an application. If the underlying scene graph is portable then moving from platform to platform can be as simple as recompiling the source code.

- Scalability: along with being able to dynamically manage the complexity of scenes to account for differences in graphics performance accross a range of machines, scene graphs also make it much easier to manage complex hardware configurations, such as clusters of graphics machines, or multiprocessor/multipipe systems. A good scene graph will allow the developer to focus on the content of the application not the lower level.

- Hierarchical modelling and inheritance: one of the most important concepts in object-oriented programming is the ability to define a behaviour or a state in a parent class so that its children can easily inherit from those properties and attributes. Scene graphs provide this feature.

5.3 Design patterns

Design patterns were first defined by Gamma, Helm, Johnson & Vlissides (1995) and have been extensively used since then. Throughout this project two behavioural patterns were used: the observer pattern and the visitor pattern.

5.3.1 The observer pattern

This pattern defines a one-to-many dependency between objects so that when the state of one object changes, all its dependents (observers) are notified and updated automatically. A typical situation when this concept is useful is described as follows:
In a small flock of two fish, how should a particular fish update its position given the fact that it solely depends on the position of the other one?
This behaviour implies that every member of the flock is dependent on each other and therefore should be notified of any change in its state. Obviously, the size of the flock should not be an issue. To solve this problem, observers delegate the responsibility for monitoring an event to a central object: subjects.
The key of the actual implementation of this pattern is to have objects (observers) that want to know when an event occurs attach themselves to another object (subjects) that is looking for an event to occur or that triggers the event itself. When the event occurs, the subject informs the observers that it has happened.

5.3.2 The visitor pattern

This pattern has been extensively used in OSG. It aims at representing an operation to be performed on elements of an object structure. Visitor lets users define a new operation without changing the classes on which it operates. Shalloway & Trott (2000) also described this pattern. A typical use of

this pattern is stated as follows:
Let us consider the scene graph described in section 5.2.2, how can we ensure that a specfic operation (an update method) will be applied to all children of the scene graph without changing the structure of the children?
The idea is to use a structure of element classes, each of which has an *accept* method that takes a *visitor* object as an argument. *Visitors* therefore have a *visit()* method for each element class. The *accept()* method of an element class calls back the *visit()* method for its class. Separate concrete *visitor* classes can then be written to perform some particular operations. Wallace (2001) later explained a method to get rid of the *accept()* method in the visitor pattern.

5.4 Description of OpenSceneGraph

OSG is an open source high performance 3D graphics toolkit made by Robert Osfield and Don Burns, used to develop applications in fields such as visual simulation, games, virtual reality, scientific visualization and modelling. Built on top of OpenGL and entirely written in C++, it runs on all Windows® platforms, OSX, GNU/Linux, IRIX, Solaris™, HP-Ux, AIX and FreeBSD operating systems. It also provides a viewer to display the 3D virtual scene. The whole graphic pipeline to render the scene is directly embedded in the core library. Every frame, the scene graph is traversed and objects in the view volume are then drawn. Figure 5.2 shows its functional components.

- The core OSG libraries provide essential scene graph and rendering features, as well as additional features required by 3D graphics applications.
- NodeKits extend functionalities of core OSG scene graph node classes to provide higher-level types and special effects such as particle systems or precipitation effects (rain, snow and fire).

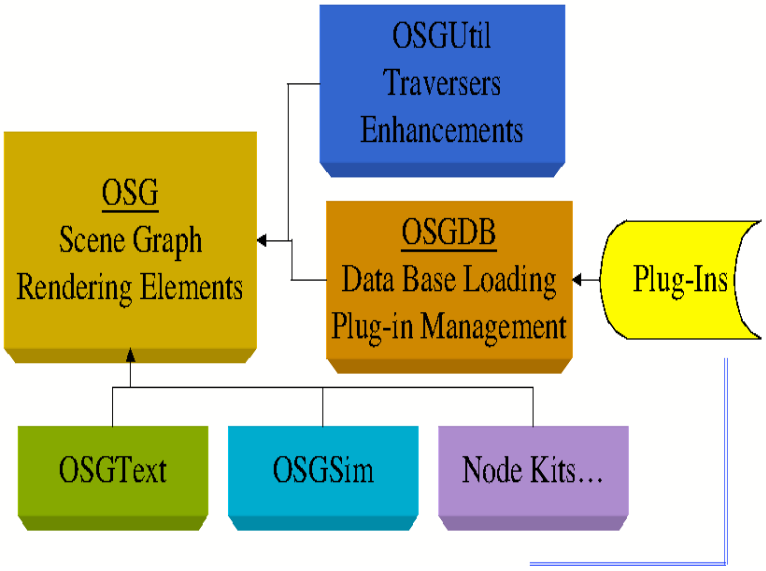


Figure 5.2: Core OSG (Martz (2007))

- There are forty-five plug-ins in the core OpenSceneGraph distribution. They offer support for reading and writing both native and third-party file formats.

5.4.1 The core OSG library

This part of OpenSceneGraph provides core features, classes and methods for operating on the scene graph. It is made up of four libraries:

- **The osg library** contains all classes related to the scene graph node. It also contains class for mathematical operations and rendering state specification and management.

- **The osgUtil library** contains classes and methods to operate on a scene graph and its contents. This library also provides methods to gather statistics of a program (**StatHandler**), optimizing the scene graph (**optimizer**). This library also provides interfaces for geometric

operations.

- **The osgDB library** contains classes and methods to create and render 3D databases. It provides plug-ins to read and write file formats supported by OSG.

5.4.2 The processing pipeline

Rendering a scene graph is achieved by traversing the graph and sending the resulting state and geometry data to the graphics card as OpenGL commands. All these happen on a frame-by-frame basis. Figure 5.3 shows the pipeline processing.

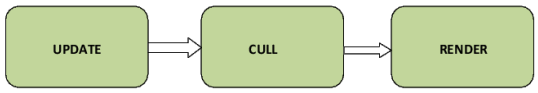


Figure 5.3: OSG pipeline processing

5.4.2.1 The update traversal

This traversal modifies the state and geometry. These updates are performed either by the application or through callbacks functions attached to nodes that they operate on. A full description of the callback mechanism will be given in the section 5.5. By updating a node, the developer changes its attributes such position and colour.

5.4.2.2 The cull traversal

After the previous traversal is completed, a cull traversal is performed in order to find what objects will actually be seen in the screen and pass a reference to the visible objects into the final rendering list. This traversal is

also in charge of ordering nodes for blending. The output of this process is the render graph.

5.4.2.3 Render traversal

This traversal uses the render graph generated by the previous traversal and sends this to the underlying hardware for rendering. OSG has a multi-processing architecture. In fact, the processing pipeline is achieved in parallel.

5.5 Dynamic modification of a scene gaph

OSG allows developers to dynamically alter the scene during the update traversal in order to create animations. Users can make use of the infinite loop to insert an update function. Recall the update traversal is fired off by the following call: *viewer.update()* while the cull and draw traversals are called by *viewer.frame()*. By using callbacks, the program is more maintenable and easier to use. Because of its multithreaded architecture, using callbacks is more efficient. Users can define their own functions (callbacks) that will be applied to the specified nodes or to any subtypes of the scene graph during the update traversal. During the update traversal, if a node is attached to a callback, OSG will call that function instead. The core library provides the *osg::NodeCallback* interface to the developer to achieve this goal. In order to define a callback, the following steps should be taken into account.

1. Derive a new class from the *osg::NodeCallback* interface provided by the core library.

2. Overload the function call operator. In fact, the dynamic alteration of the scene graph is achieved within this method.

3. Create an object from a class derived from *osg::NodeCallback* and bind it to the specified node using the *setUpdateCallback* method.

5.6 The coding of the flocking algorithm

In order to successfully implement the flocking algorithm in OSG, the following classes were used throughout the project.

5.6.1 OSG classes

1. *The Node class* is the interface of every internal node in the scene graph. It also provides methods to enable operations on the scene graph such as traversals, culling, callbacks and state management.

2. *The MatrixTransform class* uses a matrix to apply transformations on its children. The so-called matrix can be used to scale, skew, translate or rotate its children. The rotation interface of this class provides a quaternion mechanism to work out transformations, meaning that users do not have work out a separate quaternion. This is very important as this concept was extensively throughout the implementation. All these transformations are made from a relative frame. This class inherits from the *Transform* base class. This is where the callback will be attached.

3. *The PositionAttitudeTransform class*, unlike the *MatrixTransform* class, it does not used a matrix to apply transformations on its children, instead it uses a 3D vector and a quaternion to apply the translation and the rotation respectively.

4. *The Group class* is the base class for any node that can get children. It provides a parent interface. This class is actually one of the most important classes in OSG because this is where users should organize the scene graph.

Figure 5.4 shows an example of a scene graph with all these classes. It represents a small flock of three fish.

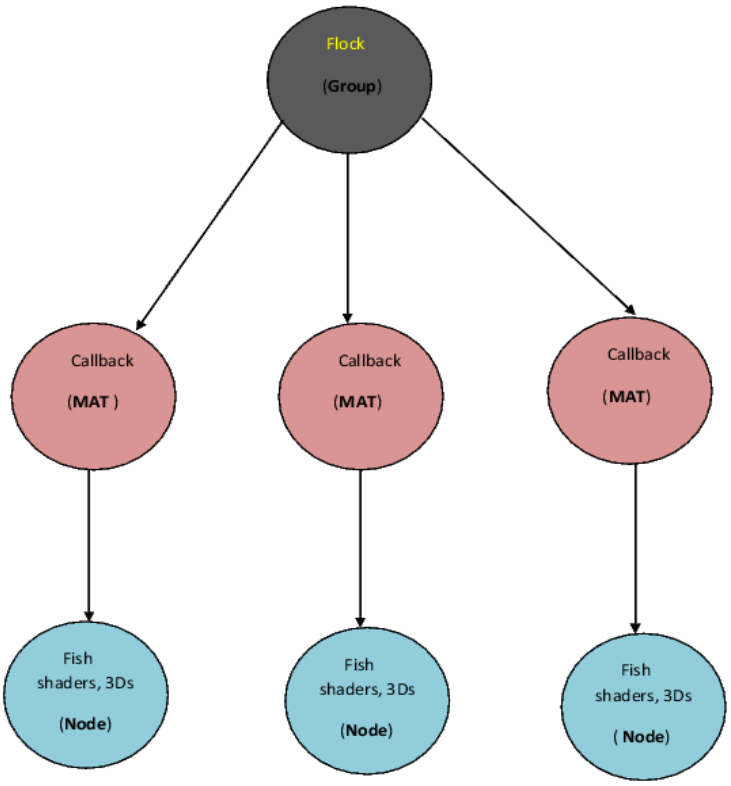


Figure 5.4: A scene graph containing a flock of three fish

5.6.2 Flocking implementation classes

Referring to the figure 5.4, the following classes were recorded:

- **Flock Manager** is the parent of every flock member, it becomes obvious that this is a *Group* object. Its main responsibilities were to maintain a list of all children and pass that list to every child as soon as it is required. At this level, no callback was attached to this node.
- **Transform Manager** provides an interface to apply transformation (translation, rotation and scaling) to fish. As described in the previous section, this class is a *MatrixTransform*.

- **Callback** stores all information related to the fish such as its position, its acceleration, its angular displacement and its velocity. This class is responsible of the applications of transformations to its children (fish) as discussed in section 5.6.1. In order to achieve this goal, it has to be attached to a node (*MatrixTransform*) which in turns has a fish as a child.

- **Fish** represents the fish objects. This is where the geometry and the attributes of the fish are stored together with its 3D model and shaders. All these attributes will be detailed in chapter 4. This is where special effects will be achieved. The class diagram of this program is shown in appendix A.

5.6.3 Utility classes

The following classes were also used in the coding process:

- **Timer** provides an easy-to-use interface to retrieve the time elapsed since the beginning of the simulation. This timer has been used to update shaders and create effects such as the fish body motion and caustics.

- **KeyboardEventHandler** also supplies an input interface to handle keyboard events. This greatly increases the interactivity of the program.

- **HUD** gives a straightforward interface to create the head-up display. It is actually a wrapper of several OpenGL functions.

Having described the system architecture, there are still some dark spots that need to be clarified. What is happening within the callback in order to have the flocking algorithm working? The answer to this question will be detailed in the next section.

5.6.4 Flocking algorithm revisited

In chapter 3, the flocking algorithm was unwound, the pseudo-code of the callback loop (actually this is the definition of the function operator) to be applied to every *MatrixTransform* node is specified as follows:

1. Request the list of all flock members from the flock manager.
2. Work out the nearest flockmates
3. Applying flocking rules described in chapter 3 to the current fish
4. Compute orientation of fish
5. Update position and rotation

There is no rendering function call in this callback, recall from figure 5.3 that this callback will be called during the update traversal. The cull and draw traversals will process the scene graph built by the update traversal.

5.7 Optimizing the scene graph

The scene graph as defined in section 5.2 can be easily optimized. The *osgUtil* library provides interfaces to traverse the scene graph in order to alter it for optimal rendering and gather statistical data. These classes are:

- *Optimizer* as its name suggests, optimizes the scene graph. Its behaviour is controlled by a set of flags indicating a specific type of information to be processed. For instance, the *FLATTEN_STATIC_TRANSFORMS* flag transforms geometry non-dynamic transform nodes, hence optimizing the rendering stage by getting rid of modifications to model-view matrix stack.
- *Statistics* and *StatsVisitor* returns the amount and types of nodes in a scene graph and the amount and types of geometry being rendered.

5.8 Major issues

5.8.1 OSG issues

The first of them is related to the lack of documentation and support in OSG making the coding process very difficult. Even the official reference guide is fragmented and incomplete. There are not many books talking about OSG. In fact, Martz (2007) wrote the first OSG book. His book does not describe in depth all OSG features. Vital parts of OSG such as shaders, particle systems are not covered in his book. Many features offered by OSG are not properly used by developers because of the lack of documentation.

5.8.2 Flocking algorithm drawbacks

The initial version of Reynolds' flocking algorithm described only three rules enough to simulate a complex behaviour. However, in order to increase the realism of the motion, additional rules should be added. Such rules include the obstacle avoidance, the predator and prey rule, the goal seeking rule and the hunger. The obstacle avoidance rule described by Reynolds (2000) was implemented in this project.
Another issue with the flocking algorithm is the fact that some rules are contradictory at times. In fact, it may happen that the collision avoidance rule and the velocity matching rule returned contradictory vectors. The way around it is to prioritize these rules. In this case, collision avoidance was first applied, followed by the velocity matching and the flock centring.
The naive implementation of the flocking algorithm suffers from a quadratic bottleneck meaning that as the size of the flock increases so is the computational time requires to apply the flocking motion. Reynolds suggested the use of spatial hashing in order to overcome this issue, however this was not covered in this project.

CHAPTER

SIX

Software evaluation

6.1 Evaluation of the software

Having described in depth the analysis and coding process in previous chapters, this chapter will mainly focus on presenting results. The software was tested on an AMD Turion^{TM} 64 Dual Core running at 1.66 GHz. The graphics card used was a PCI Express nVIDIA GeForce^{TM} 7600 with 256 Mb of Video RAM with a core speed of 560 Mega Hertz. The memory bandwidth is 22.4 Gb per second.

6.1.1 Results

The 3D models of the fish and the terrain were obtained from Toucan (2007) and RenderMonkey^{TM} test suite respectively.

6.1.1.1 Fish shoaling

Figure 6.7 shows three flock of thirty four fish, while Figure 6.1 shows a school of fish in Pixar's *Finding Nemo*.

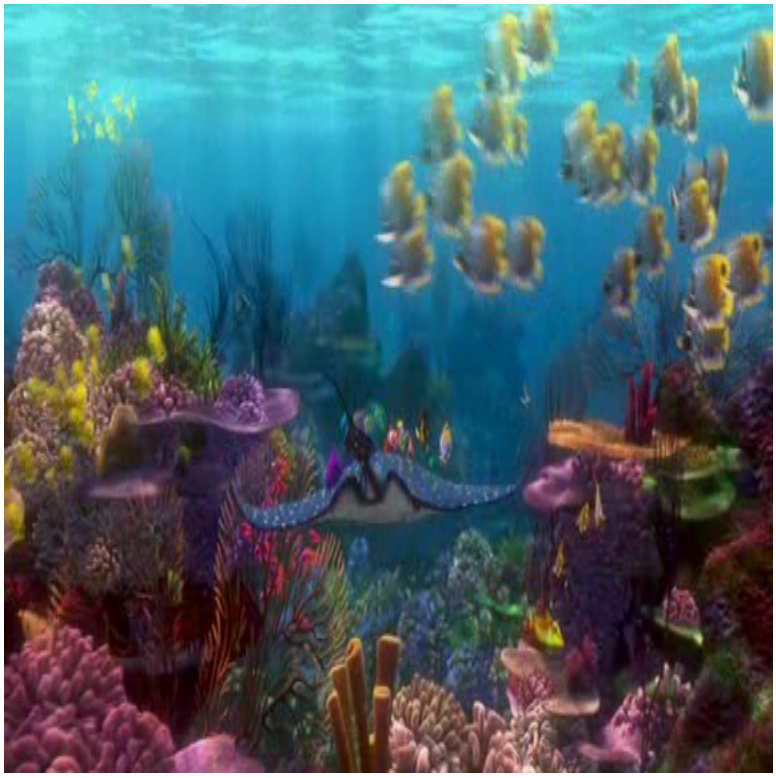


Figure 6.1: A school of fish (*Pixar's Finding Nemo*)

6.1.1.2 Obstacle avoidance

6.1.1.3 Terrain visualization

Figure 6.3 shows the terrain visualization. An overview of the overall rendering will be presented in section 6.1.1.5. The fogging effect and the bubbles will also be shown in section 6.1.1.5

6.1.1.4 Caustics

Figure 6.4 shows a screenshot of the caustics effect achieved within the project. It was implemented using the map shown at the figure 6.5. Figure 6.6 shows the caustics effect in *Finding Nemo*.

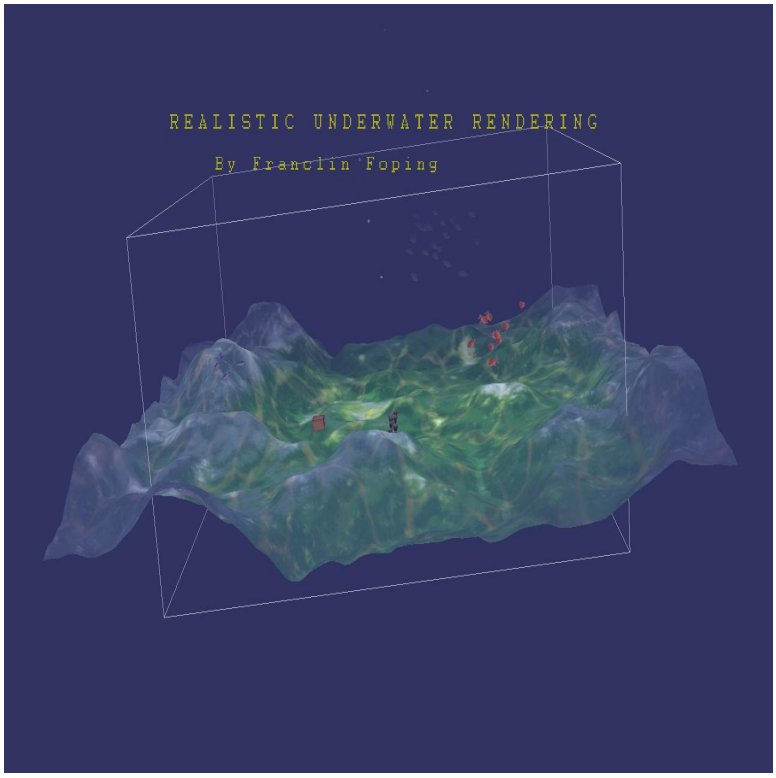


Figure 6.2: The obstacle avoidance rule

6.1.1.5 Underwater visualization

Figure 6.7 shows the final rendering of the software, users can interact with the objects using the mouse and the keyboard. Furthermore, items such as caustics, grass, bubbles and fish schooling can be seen.

6.2 Performance of the program

The overall performance of the simulation is a multidimensional quantity. There are separate costs for large simulations as well as other costs related to the animation of every fish and the terrain. All these use a vertex and a fragment shaders. In OSG, there is a clear bottleneck during the draw traver-

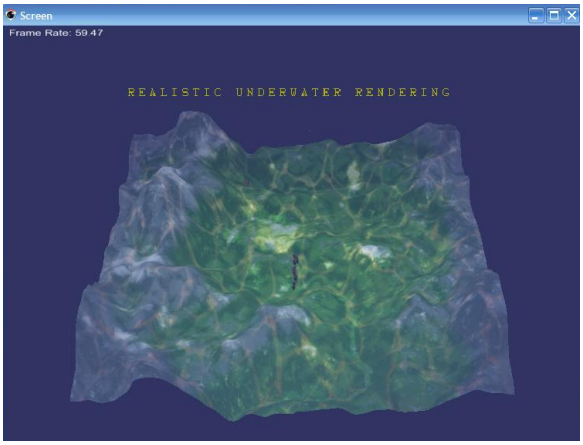


Figure 6.3: The terrain

Figure 6.4: Caustics effect

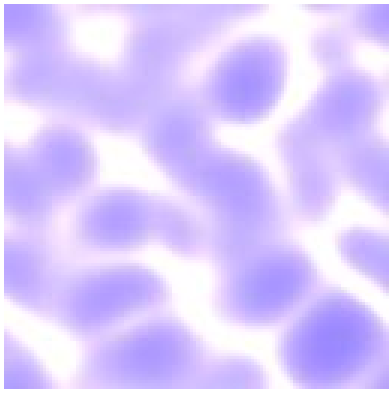


Figure 6.5: Caustics map (Stam (1996))

sal because the scene consists of more than one hundred and fifty thousand vertices.

The software developed throughout this project can simulate fifty fish at a frame rate of 60 on an nVIDIA GeForce^{TM} 7600 graphics card. This rate includes the caustics effect, bubbles, fog and the fish body motion.

However, as the population grows up to 1000, the frame rate drops to 50 when the graphics effects are turned on and 60 when turned off. The main reason of this gap is the quadratic bottleneck of the Reynolds' flocking al-

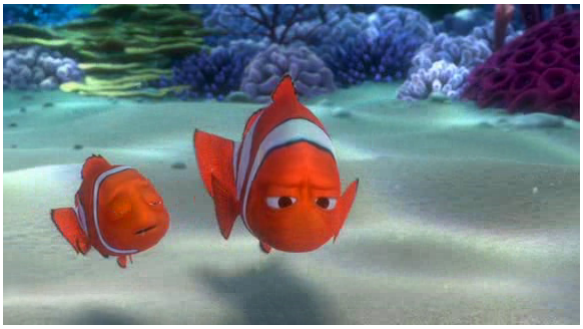


Figure 6.6: Caustics effect (Pixar (2003))

gorithm. Throughout this project, spatial hashing was not used in order to optimize the algorithm.

The animated fish bodies used in the software have complex geometries containing more than three thousand vertices. Their bodies have a swimming motion in the tail provided by a travelling sine wave function in their OpenGL Shader Language (GLSL) vertex shaders.

Figure 6.8 shows a screenshot of the statistics of the software. This is the result of pressing the 's' four times. The viewer of the application (*osgViewer*) first displays the frame rate, then the time spent during each traversal (update, cull and draw). Then it displays information about the geometry of the scene processed every frame. In this case, the application is synchronized with a 60 Hz monitor. It is important to point out that the time spent in the update traversal (0.35 millisecond) is so insignificant that the viewer does not display it graphically. Recall from section 5.4 that every callback is executed at this level. The outstanding optimization technique described in chapter 4 can be praised here. However, the cull and draw traversals display as cyan and dark yellow in the graphical display. The cull traversal takes 1.04 millisecond while the draw traversal only takes 4.25 with all shaders being taken into account. All the geometry of the scene is rendered in 3.58 milliseconds by the GPU, this time is shown in red. Finally, the viewer processed 292 drawable objects, 167,390 vertices and 291,916 triangles.

6.3 Comparison with other implementations

Boids took about one hour to simulate one second of flocking animation of 80 boids with a frame rate of 30. The simulation was performed on a 1 MHz CPU and was an off-line process. Using spatial hashing, Reynolds (2000) described an interactive simulation with 280 boids running at a frame rate of 60 on a PlayStation®2.

Treuille, Cooper & Popovic (2006) described a crowd model which is a unique hybrid of a fluid and crowd models. A continuum field is created every frame to globally characterize the crowd and the environment, then individual agents navigate according to this field. On a fast computer, it can run a simulation with 10,000 agents at 5 frames per second without graphics. The simulation rate is 2 with graphics, but that provides a thread displaying interpolated frames showing humanoid characters at 12 frames per second.

Tecchia, Loscos, Conroy & Chrysanthou (2001) described a system in which every member of the simulated crowd reacts to each other albeit with a fairly simple behavioural model. The system ran on a single processor with 5000 individuals at 37 frames per second and with 10,000 individuals at a frame rate of 21. These rates include the load of rendering an urban setting and animated humanoid representations of each individual.

Shao & Terzopoulos (2005) described an autonomous pedestrian model that exhibited both high performance and sophisticated goal-driven models of people at a train station. Without graphics their system can simulate 1400 pedestrians at a frame rate of 30 on a modern computer. With humanoid character animation and rendering of the complex environment results in rates of 3.8 frames per second for 500 individuals.

The FastCrowd system described by Courty & Musse (2005) ran a crowd of 5000 agents at about 100 frames per second, and a crowd of 10,000 at 35 frames per second (excluding visualization, 50 frames per second and 20 frames per second with agents rendered as 2D disks).

The GPU-based Boids described by Erra et al. (2004) could simulate a flock

of 1600 boids at 60 frames per second, with 8000 boids the system ran at about 20 frames per second. These rates include rendering a 3D scene with animated bird models.

Quinn, Metoyer & Hunter-Zaworski (2003) used a distributed multiprocessors to run a large evacuation scenarios in which 10,000 pedestrians are simulated with a frame rate of 45 on ten processors of the swarm cluster is connected by a gigabit Ethernet switch.

Reynolds (2006) described a simulation for the PlayStation®3 in which 5000 high detail fish at 30 frames per second with level of detail, underwater haze, animated water surface and high dynamic range illumination. Figure 6.9 shows his result.

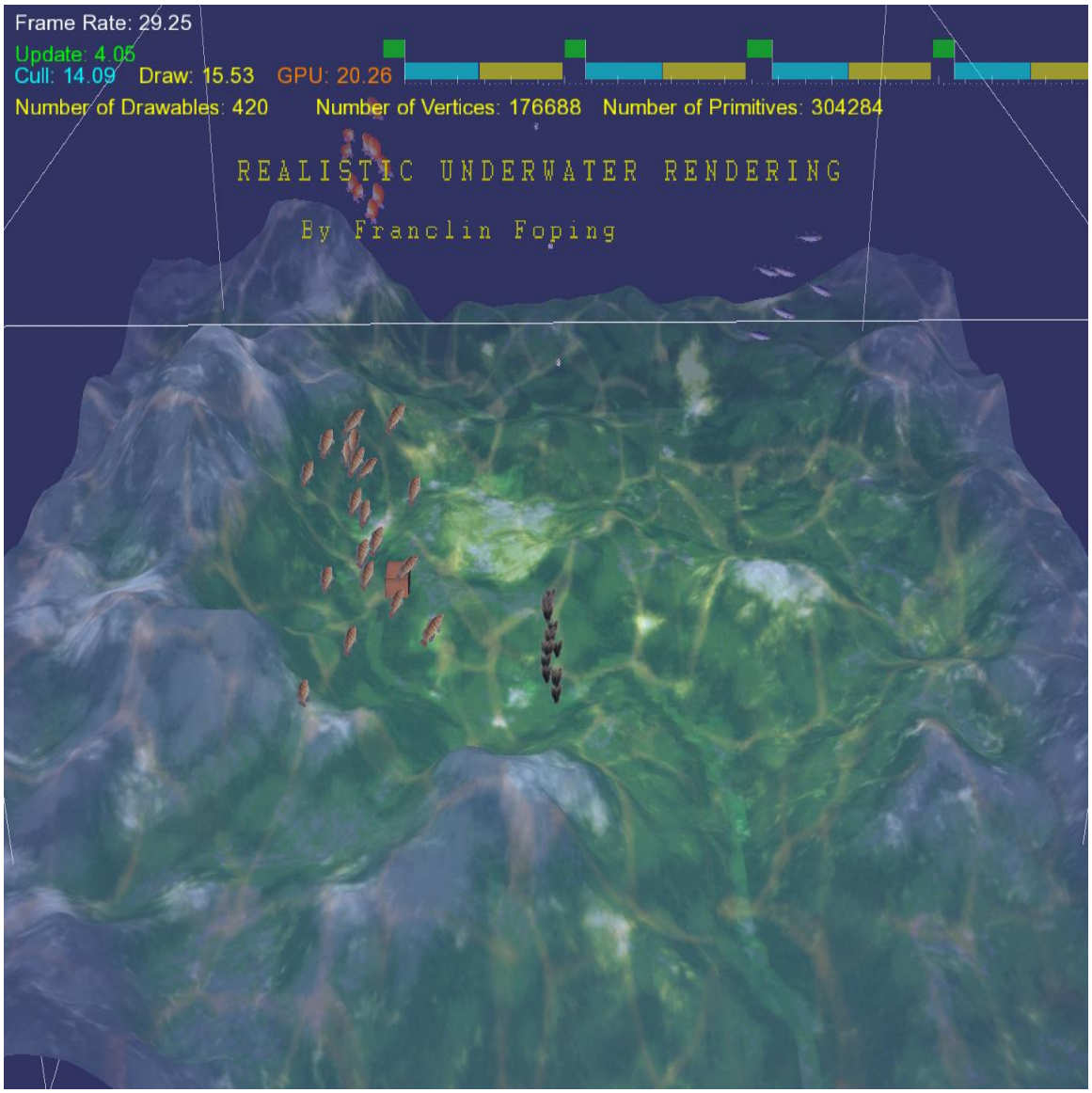


Figure 6.7: The final rendering

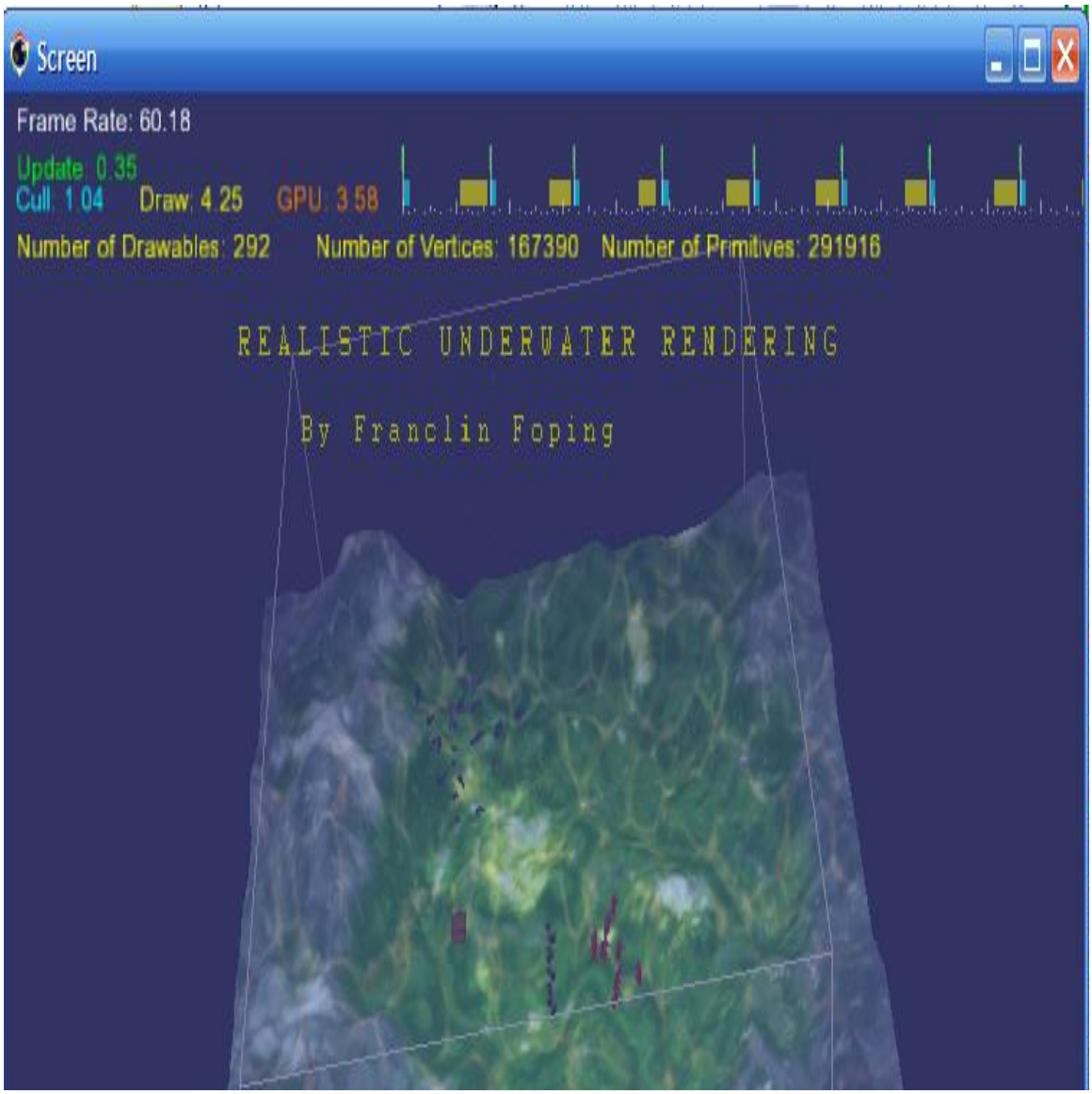


Figure 6.8: Statistics of the simulation

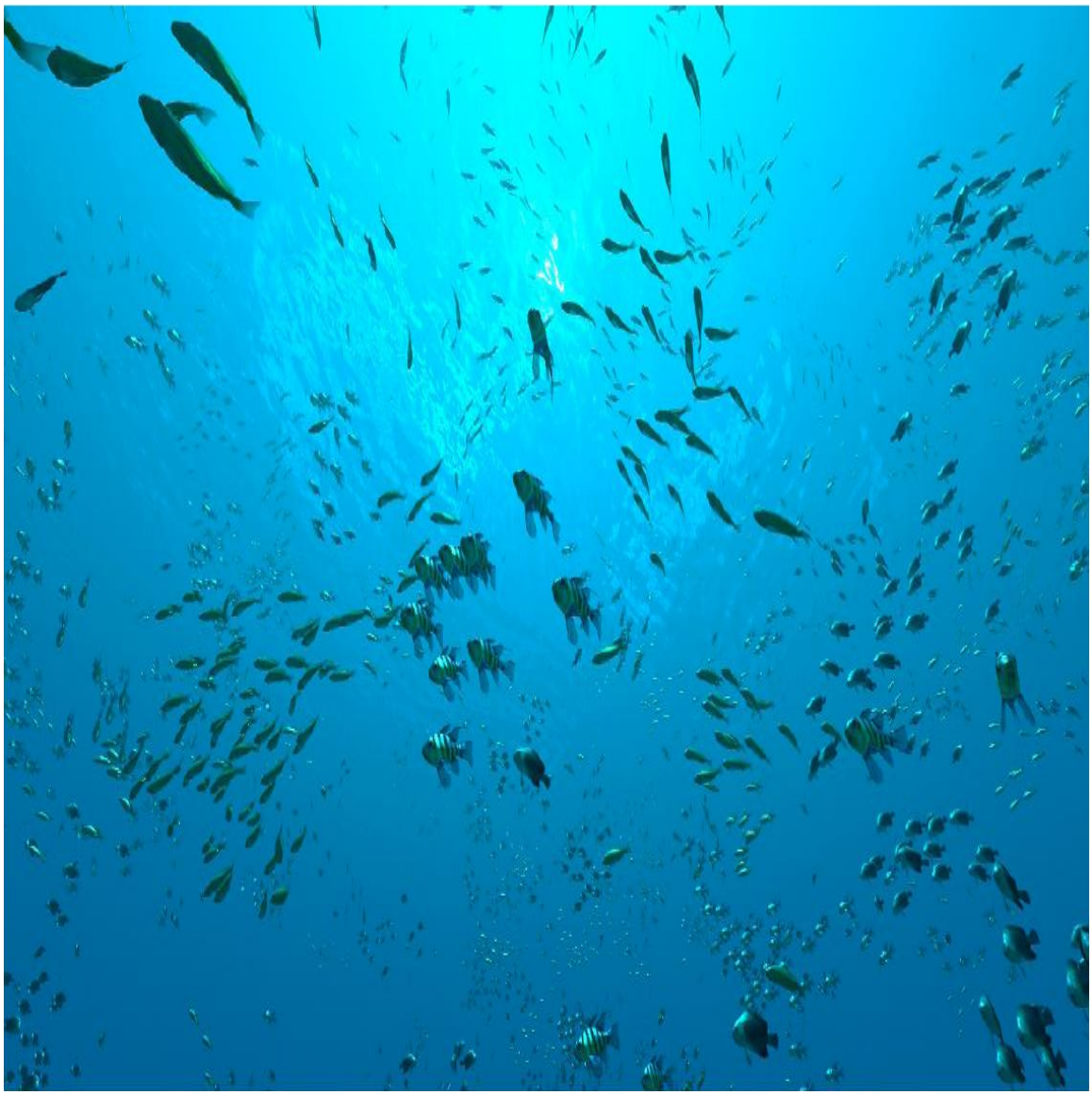


Figure 6.9: A school of fish (Reynolds (2006))

CHAPTER

SEVEN

Critical appraisal

7.1 Objectives summary

All the project objectives, mentioned in chapter 1.3 were realised in the final version of the software. The simulation runs at over 38 frames per second. The fish shoaling effect seems impressive and will be used for further investigations into the VENUS project. However, if the project was completed on time, everyone would have been pleased.

Being able to learn OSG in such a short amount of time was one of the most exciting parts of the project. Using programs like 3Ds Max, RenderMonkey was also other skills gained from the project. As a graphics programmer, it is really important to learn more about shaders and GPU programming. This project gives me the opportunity to learn in depth the graphics pipeline and use my mathematics skills.

The project starts with just a rough idea and after carefully on working on fish shoaling, the finish product is something completely amazing! Extra items have been added to the project, due to my motivation to do achieve a world-class project. Caustics, bubbles, fog and grass were added as part as my personal motivation. The obstacle avoidance was also a plus. What

about the fish body motion using shaders?

7.2 Project management

Generally speaking, the project management was excellent. Although, I faced some issues due to the lack of documentation in OSG and my health which were not really helpful.
Every milestone was carefully reached and completed on time. The result of all these is a world-class project that will be used in an archaeology project (VENUS).

7.3 Lessons learnt

Upon successful completion of the project, a lot has been mastered.

- **Design patterns**: as mentioned in chapter 5, OSG is built on top of C++ and designers have used design patterns a lot. Throughout the project a successful combination of the design pattern theories and practice in C++ and OSG was achieved. The callback, observer and visitors all fall in that category.

- **Advanced C++ skills**: this project would have not been completed if advanced C++ skills were not mastered in a short amount of time. Using advanced standard template library (STL) idea such as function objects was of great help.

- The main part of the project relies only on the good understanding of OSG, all that in three months without documentation and support. Even Robert Osfield would be stunned to hear that.

- Another important part of the project was the learning of 3D Max in order to edit 3D models.

- Being able to do GPU programming from scratch was another positive aspect of the project. Many lessons were learnt from that unprecedented experience.

- Like any multithreaded software, there are lots of issues that have to be handled. Although OSG provides the mult-threaded interface to deal with the operating system, developers are responsible of synchronizing their objects. Recall from section 5.4 that the draw process in OSG is made up of two sub-processes: the dispatch to the CPU and the actual rendering on the GPU. However, because of its parallel architecture, OSG only ensures that the draw traversal only returns after processing all static data. If the data is dynamic, threads may collide and the application is likely to be halted by the operating system. Developers are responsible of specifying the right state of their objects by using *data variance*. Again the poor documentation of OSG was not really helpful.

CHAPTER

EIGHT

Conclusion and future works

The project itself aimed at applying Reynolds' flocking algorithm to 3D fish models. The realism of the scene was enhanced by adding bubbles, grass and an underwater terrain. The rules of flocking include the collision avoidance which states that every fish should steer to avoid local flockmates, the velocity matching by which a given fish should match the velocity of its nearest flockmate and finally the flock centring which urges every fish to always head toward the centre of the flock.

By using callbacks, it is possible to implement these rules in OSG. Although poorly documented, OSG is one of the most powerful graphics toolkit available. Completely free of charge and open-source, its architecture was presented in chapter 5. The implementation of the three simple rules in OSG brings about a complex behaviour that is seen everyday: the motion of an aggregation of fish.

Further works should first focus on increasing the realism of the scene. The caustics effect achieved within this project was a fake for it does not take into account the incoming light from the surface.

In order to have a realistic school of fish, some additional rules must be taken into account. However, the obstacle avoidance under which the entire flock

always steer in order to avoid any obstacles in the world was successfully added. Furthermore, some flockmates can be hungry, in that case they have to leave the flock in order to find food. There may be some predators flying around, the entire flock should then steer away from predators, hence the need to add another rule: the predator avoidance rule. Other rules involve the mating rule, where within the flock, males will always try to find the nearest female available.

The naive implementation of this algorithm suffers from a quadratic bottleneck meaning that as the size of the flock increases, the computational time becomes more and more important. In fact, as described in chapter 6.1, the performance of the software also relies on the costs of animating every fish. Recall from the chapter 4 that the overall simulation contains more than one hundred and fifty thousand vertices. There are vertex and fragment shaders attached to every node of the scene graph. In order to improve the algorithm, spatial hashing techniques should be used. In fact, Bell, Yu & Mucha (2005) noticed that the use of spatial hashing has become nearly ubiquitous for flock modelling as well as granular models of physical phenomena. Implementing spatial hashing in 3D requires the uses of octrees while in 2D, one should use quad-trees. Because of time limitations, all these techniques were not implemented throughout this project.

Fuzzy logic algorithms should be applied on the fish in order to allow them to find their nearest flockmates in a more intuitive way. This will greatly increase their ability to sense nearby fish. Foping (2006) demonstrated that the problem of finding a set of a random number of nearest neighbours is *NP-Complete* and is therefore among the most difficult problems in computer science. Indeed this problem is equivalent to the traveling salesman problem which can be stated as follows: *Given a number of cities and the costs of traveling from a city to another one, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?*

APPENDIX

A

The flocking implementation class diagram

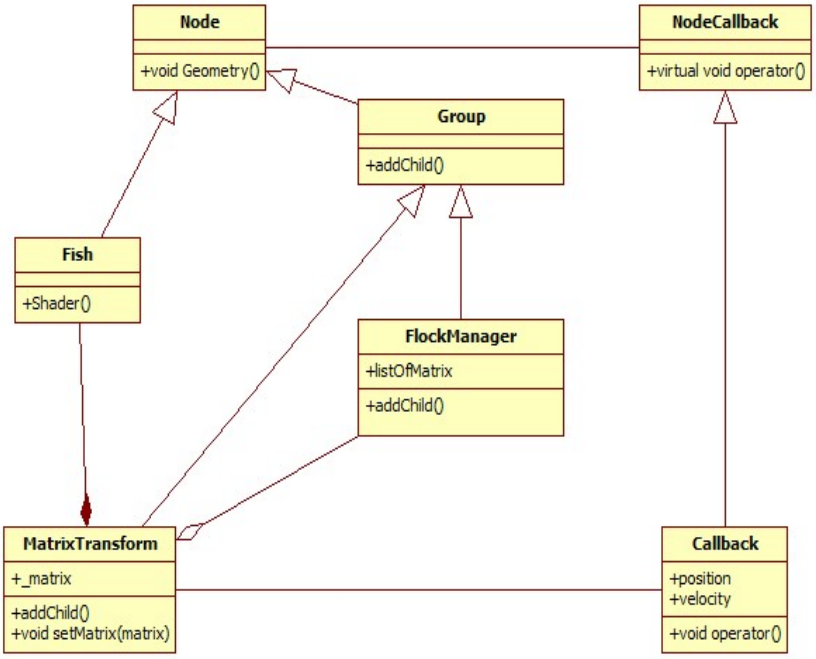


Figure A.1: The simplified class diagram

APPENDIX

B

The scene graph sketch

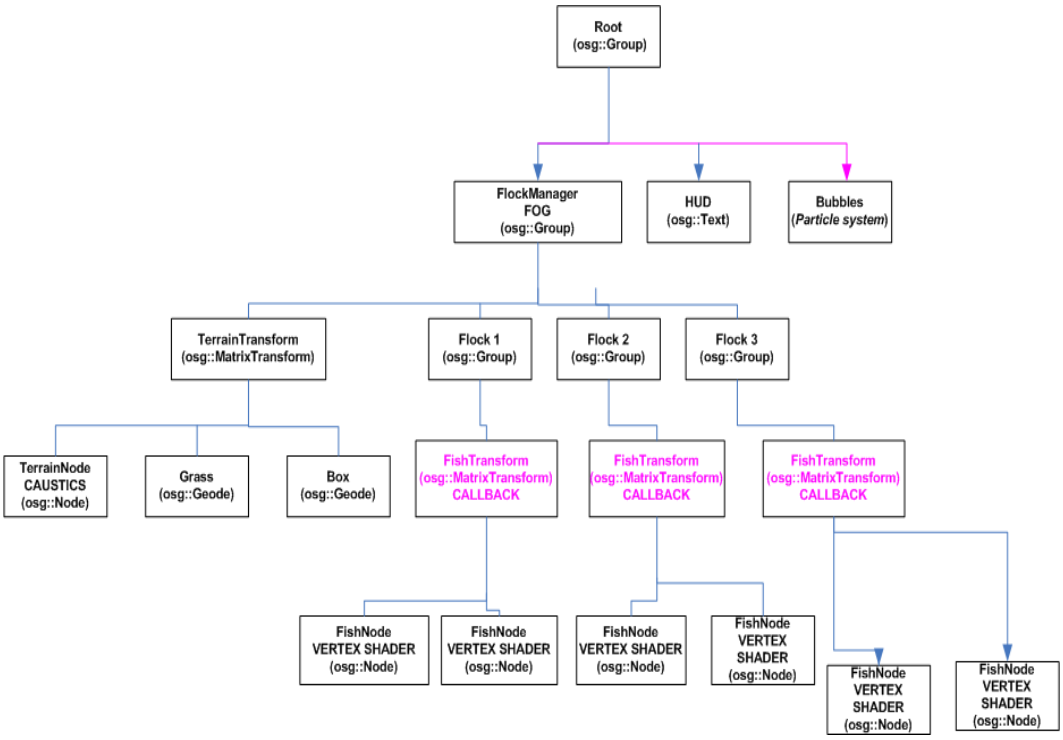


Figure B.1: The scene graph sketch

APPENDIX

C

User manual

In order to successfully run or compile the software, some hardwares and softwares are required.

C.1 Requirements

C.1.1 Software

To compile the software, users should install the integrated development environment (IDE), Microsoft Visual Studio^{TM} 2005 with the latest updates available on the Microsoft website. In addition of installing Visual Studio 2005, users should also install OpenSceneGraph 1.9 available for free at the website (`http://www.openscenegraph.org`). The operating system used in this project is Microsoft Windows® XP SP2.

C.1.2 Hardware

As mentioned in chapter 5, the software makes use of shaders to animate the fish body, create the fog effect and the caustics effect. Both vertex and fragment shaders were successfully tested on nVIDIA® graphics cards

namely the GeForce^{TM} 7600 and 8600. No tests were made on any other cards. Furthermore, the multithreaded architecture of OpenSceneGraph was described in chapter 5, therefore a dual core CPU should be used for better performance. The software was successfully tested on an AMD Turion® 64 X2 and Athlon® 64 X2 CPUs. As previously, no tests were performed on Intel®'s dual core CPUs.

C.2 Installing the software

On the CD-ROM, the folder named SETUP contains a Microsoft® Installer (MSI) file that can be used to install the software on a Windows® plateform. By double-clicking on it and following the instructions, the software should be immediately ready for a full use. Now, open the folder you have just chosen and double on the executable file (with a fish icon) and **voila**!

C.2.1 Controls

C.2.1.1 Mouse

By maintaining the left button of the mouse and move it, users can rotate the world.
Users can zoom in and out the world by maintaining the right button of the mouse and move it. This is useful to see the fog effect in action. Check this out!

C.2.1.2 Keyboard

- The 'S' key displays the statistics in real-time of the simulation. By pressing it many times, different statistics are displayed. Warning, on some GPUs, this may freeze the simulation.
- The 'W' key displays the wireframe mode of the simulation
- The 'Echap' key quits the program.

- The 'F' key toggles the full screen mode.
- The 'b' key toggles the backface culling on the bounding box depicting the world.
- The space bar positions the camera at the centre of the world. This is very useful.
- The 't' key disables/enables textures on grass and box.
- The left arrow key scatters the flock of perch. The right arrow key enables the flocking behaviour on them.
- The left Control key scatters the flock of tunas. The right Control key enables the flocking behaviour on them.
- The left Shift key scatters the flock of lionhead. The right Shift key enables the behaviour on them.

C.3 Compiling the source code

On the CD-ROM, the folder name SOURCE contains the solution file for Visual Studio^{TM} , however some environment variables must be defined and set to the correct values before compiling the source code. These environment variables are:

- *OSG_INCLUDE_PATH* which is a string defining the name of the directory containing all the included files of OSG.
- *OSG_LIB_PATH* which is also a string defining the name of directory containing all the library files of OSG

CAUTION

- The compilation cannot be successful if the **Multi-threaded Debug DLL (/MDd)** option is set in Visual Studio. This option is the default option if the CPU has a dual core architecture. This option can be

set in Project property (C/C++ – Code Generation – Runtime library).

- The Run-time Type identification (RTTI) must be enabled in Visual Studio. This can also be done in project property. Under the C/C++ tab, select C++ language in the category box. There is a checkbox right there for 'Enable RTTI' make sure you have it selected and ticked.

C.4 Recommended configuration

The following configuration is highly recommended to run the software:
Main Memory: 2Gb
CPU: AMD Turion 64 or Athlon® Dual Core
GPU: nVIDIA GeForce™ 7600 or 8600
OpenGL version 2
Operating system: Microsoft Windows XP™ Service pack 2
Color quality: 32 bits

REFERENCES

Aranha, M. (2005), Realistic Underwater Visualisation, Computer Graphics Group, University of Bristol.

Bell, N., Yu, Y. & Mucha, P. (2005), Particle-based simulation of graular materials, *in* 'Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on computer animation', ACM Press, pp. 77–86.

Blackwell, T. M. (2005), Swarm Music: Improvised Music with Multi-Swarms, *in* 'Proceedings of the 2003 AISB Symposium on Artificial Intelligence and Creativity in Arts and Science', AISBJ, pp. 41–49.

Briere, N. & Poulin, P. (2001), 'Adaptive Representation of Specular Light Flux', *Computer Graphics Forum* **20**(2), 149–159.

Carlson, D. & Hodgins, J. (1997), Simulation Levels of Detail for Real-time Animation, *in* 'Proceedings of Graphics Interface '97', pp. 1–8.

Chapman, P., Conte, G., Drap, P., Gambogi, P., Gauch, F., Hanke, K., Longand, L., Loureiro, V., Papini, O., Pascoal, A., Richards, J. & Roussel, D. (2006), VENUS: Virtual ExploratioN of Underwater Sites, *in* '7th International Symposium on Virtual Reality : Archaeology and Cultural Heritage', VAST.

Chapman, P., Viant, W. & Munoko, M. (2004), 'Constructing immersive environments for the visualization of underwater archaeological sites', *Computer Applications and Quantitative Methods to Archaelogy Conference*.

Courty, N. & Musse, S. (2005), Simulation of large crowds in emergency situations including gaseous phenomena, *in* 'Proceedings of IEEE Computer Graphics International', pp. 206–212.

Deoswitz, B. (2003), Computer graphics world, preprint.

Deusen, O., Ebert, D. S., Fedkiw, R., Musgrave, F. K., Prusinkiewicz, P., Roble, D., Stam, J. & Tessendorf, J. (2004), The elements of nature: interactive and realistic techniques, *in* 'SIGGRAPH : '04: ACM SIGGRAPH 2004 Course Notes', ACM Press, p. 32.

Dunn, F. & Parberry, I. (2002), *3D Math Primer for Graphics and Game Development*, Wordware Publishing Inc.

Erra, U., Chiara, R. D., Scarano, V. & Tatafiore, M. (2004), Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance, *in* '9th International Fall Workshop on Vision, Modeling and Visualization'.

Foping, S. F. (2006), A polynomial algorithm to find the reversal degree of planar graphs, Master's thesis. University of Yaounde I, Cameroon.

Gabbai, J. M. E. (2005), Complexity and the Aerospace Industry: Understanding Emergence by Relating Structure to Performance using Multi-Agent Systems, PhD thesis. University of Manchester.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gates, W. F. (2002), Animation of Fish Swimming, Technical report.

Heppner, F. (1987), A stochastic nonlinear model for coordinated bird flocks, American Ornithological Union Meeting (abstract) San Francisco.

Heppner, F. & Grenander, U. (1990), A stochastic nonlinear model for coordinated bird flocks, *in* 'The Ubiquity of Chaos, AAAS, Washington', pp. 233–238.

Iwasaki, K., Dobashi, Y. & Nishita, T. (2002), 'An efficient method for rendering underwater effects using graphics hardware', *Computer Graphics forum* **8**, 1–11.

Kelly, K. (1995), *Out of control: the new biology of machines, social systems and the economic world*, Perseus Books Group.

Lander, J. (1998), 'Better 3D : The Writing Is on the Wall', *Game Developer* **1**, 15–21.

Martz, P. (2007), *OpenSceneGraph Quick Start Guide*, Skew Matrix Software.

Okubo, A. (1986), 'Dynamical aspects of animal grouping: swarms, schools, flocks and herds.', *Advances in Biophysics* **22**, 1–94.

Pixar (2003), 'Finding Nemo', `http://www.pixar.com/featurefilms/nemo/`, Accessed on 1 September 2007.

Quan, X., Zhong, S., Wang, W., Xu, Q., Liang, Y. & Zhang, Q. (2006), Modeling of Artificial Life Based Virtual Fish Behaviour, *in* '16th International Conference on Artificial Reality and Telexistence–Workshops(ICAT'06)', pp. 213–216.

Quinn, M. J., Metoyer, R. & Hunter-Zaworski, K. (2003), Parallel Implementation of the Social Forces Model, *in* 'Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics', ACM Press, pp. 63–74.

Reeves, W. T. (1983), Particle systems : A technique for modelling a class of fuzzy objects, *in* 'SIGGRAPH '83: Proceedings of the 10th annual conference on computer graphics and interactive techniques', ACM Press, pp. 359–375.

Reynolds, C. W. (1987), Flocks, Herds and Schools: A distributed behavioral model, *in* 'SIGGRAPH '87: Proceedings of the 14th annual conference on computer graphics and interactive techniques', ACM Press, pp. 25–34.

Reynolds, C. W. (2000), Interaction with Groups of Autonomous Characters, *in* 'Proceedings of the Game Developer Conference', CMP Game Media Group, pp. 449–460.

Reynolds, C. W. (2006), Big fast crowds on PS3, *in* 'Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames', ACM Press, pp. 113–121.

Shalloway, A. & Trott, R. J. (2000), *Design Patterns Explained: A New Perspective on Object Oriented Design*, Addison-Wesley.

Shao, W. & Terzopoulos, D. (2005), Autonomous pedestrians, *in* 'Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation', ACM Press, pp. 19–28.

Shinya, M., Saito, T. & Takahashi, T. (1989), Rendering Techniques for Transparent Objects, *in* 'Proceedings of Graphics Interface '89', pp. 173–181.

Stam, J. (1996), Random caustics: natural textures and wave theory revisited, *in* 'SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96', ACM Press, p. 150.

Tecchia, F., Loscos, C., Conroy, R. & Chrysanthou, Y. (2001), Agent behaviour simulator (ABS): A platform for urban behaviour development, *in* 'Games Technology', GTEC.

Toucan, C. (2007), '3D Computer Graphics', [Available online: `http://toucan.web.infoseek.co.jp/3DCGE.html`].

Treuille, A., Cooper, S. & Popovic, Z. (2006), Continuum Crowd, *in* 'Proceedings of SIGGRAPH 2006', ACM Trans.

Tu, X. & Terzopoulos, D. (1994), Perceptual modelling for the behavioral animation of fishes, *in* 'Pacific Graphics '94: Proceeding of the second Pacific conference on Fundamentals of computer graphics', World Scientific Publishing Co., Inc, pp. 185–200.

Wallace, B. (2001), 'Eliminate accept() methods from your Visitor pattern', *JavaPro Magazine* **1**, 247–254.

Wand, M. & Strasser, W. (2003), 'Real-Time Caustics', *EUROGRAPHICS (3)* **22**, 1–10.

Woodcock, S. (2000), *Game Programming Gems*, Charles River Media.

Woodcock, S. (2001), *Game Programming Gems 2*, Charles River Media.