A FRAMEWORK TO INVESTIGATE BEHAVIOURAL MODELS

Leandro Motta Barros

Tatiana Figueiredo Evers

Soraia Raupp Musse

Centro de Ciências Exatas e Tecnológicas
UNISINOS - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950, São Leopoldo, RS
Brazil
{lmb, tatiana, soraiarm}@exatas.unisinos.br

ABSTRACT

This paper presents a framework to investigate behavioural models of multiple agents. The framework is designed to deal with problems commonly found in behavioural animation systems, most of which are caused by excessive coupling between the system modules. Hence, the framework is designed to be modular, flexible and extensible. Behavioural animation models based on this framework are clearly divided in modules that can be independently designed and developed. Furthermore, these modules can be easily substituted, modified and reused. We present modules related to crowd behaviour, intelligent camera, and virtual environment and how these are integrated using the Python programming language. We also discuss visualization aspects, which are addressed by yet another module.

Keywords: behavioural animation, autonomous agents, inteligent camera

1 INTRODUCTION

Current animation systems encompasses aspects from computer graphics and artificial intelligence [Aylet01]. It is often desired to develop systems that are visually and behaviourally accurate. Furthermore, the behavioural part of these systems is commonly composed of several cooperating modules.

Implemented behavioural animation systems habitually suffer from excessive coupling between their modules. This leads to some undesirable consequences:

1. Difficulty in developing the modules separately.
2. Limited flexibility, because it is hard to modify one module without having to change other modules.
3. Restricted extensibility, as the addition of new modules is not taken into account in the system design.
4. The resulting systems tend to have a noticeable emphasis either in the artificial intelligence or computer graphics aspects, neglecting the importance of the other half of the system.

In this paper we describe an open framework to investigate behavioural models of multiple agents that deals with these problems.

In the next section we review previous work. Then we describe the organization of our framework. Section 4 presents topics related to visualization. And finally, conclusions are given in Section 5.

2 RELATED WORK

Previous research on behavioural modeling has mainly focused on various methods for providing virtual agents endowed with levels of autonomy. Reynolds [Reyno87][Reyno99] described a distributed behaviour model for simulating flocks formed by actors endowed with perception skills. The Helbing's team [Treib99] describes methods to simulate the movement of pedestrians. Mataric [Matar94] and Noser [Noser96] developed local rules for controlling collective behaviours. Tu and Terzopoulos [Tu94] have worked on behavioural animation for creating artificial life, where virtual agents are endowed with synthetic vision and perception of the environment. Blumberg [Blumb95] presented the problem of building autonomous animated creatures for interactive virtual environments, which are also capable of being directed at multiple levels, and Perlin [Perli96] describes *Improv*, a system for scripting interactive actors. Recently, some behavioural systems using classifier systems [Sanza01] and petri nets [Bicho01] are presented in order to provide autonomous behaviours to virtual agents. Kallmann *et. al.* [Kallm00] described *ACE*, a common environment for simulating virtual human agents.

Our framework describes an open architecture which enables the user to plug in different interdependent behavioural clients managed by the server. We use Python to integrate the scripts handled by the modules and in this paper we present the integration with three modules: crowd behaviour, intelligent camera and virtual environment.

3 FRAMEWORK

The ideal system should be flexible, extensible and efficient. Taking into account these goals, our model is divided into independent modules written in an object-oriented language. This separation helps to focus each module in its function. Furthermore, it is then easier to test, maintain and extend code. The next section describes all modules and the overall system architecture.

3.1 Behavioural Modules

The modules are:

1. Intelligent Camera (*IC*)

The *IC* module is responsible for the automatic location of camera during the simulation. In fact, the user can define the events he/she is interested to observe. *IC* is a very useful tool for example, when there are multiple agents in a dynamic environment, lots of events can trigger simultaneously or not, increasing the difficulty to visualize them. The *IC* is able to search for events specified according to a specific syntax, as shown in Table 1.

Entity	Task	Entity
agent < *id* >	near	group
agent	far	agent < *id* >
group < *id* >	near	< *place* >
group	far	group
agent	near	agent
group < *id* >	far	group
agent	near	< *place* >

Table 1: Events syntax

For instance, if a specific agent (<id> = BOB) is near to the RESTAURANT (<place>), then the camera will be positioned near to BOB. If the triggered event concerns a movement (*e.g.* if BOB is walking) then the camera will follow him. The *IC* behaviour of following or just stay fixed in a specific place is emergent as a function of the camera perception of triggered events.

2. Crowd (*CROWD*)

This section is related to *ViCrowd* [Musse01],

a model for simulating crowds of humans in real-time. In this model, we deal with a hierarchy composed of virtual crowds, groups and individuals. The groups are the most complex structure that can be controlled with different degrees of autonomy. This autonomy refers to the extent to which the virtual agents are independent of the user intervention and also the amount of information needed to simulate crowds. Thus, depending on the complexity of the simulation, simple behaviours can be sufficient to simulate crowds. Otherwise, more complicated behavioural rules can be necessary, and in this case, they can be included in the simulation data in order to improve the realism of the animation. We present three different ways for controlling crowd behaviours: *i*) by using innate and scripted behaviours; *ii*) by defining behavioural rules, using events and reactions, and, *iii*) by providing an external control to guide crowd behaviours in real time. The two main contributions of the *ViCrowd* model are: the possibility of increasing the complexity of group/agent behaviours according to the problem to be simulated, and the hierarchical structure based on groups to compose a crowd.

At a lower level, the individuals have a repertoire of basic behaviours that we call *innate behaviours*. An innate behaviour is defined as an "inborn" way to behave. Examples of individual innate behaviours are goal seeking behaviour, the ability to follow scripted or guided events/reactions, the way trajectories are processed and collisions avoided.

While the innate behaviours are included in the model, the specification of scripted behaviours is done by means of a script language. The groups of virtual agents whom we call <*programmed groups*> apply the scripted behaviours and do not need user intervention during simulation. Using the script language, the user can directly specify the crowd or group behaviours. In the first case, the system automatically distributes the crowd behaviours among the existing groups. Moreover, externally controlled groups, <*guided groups*>, no longer obey their scripted behaviour, but act according to the external specification [Musse98].

Events and reactions have been used to represent behavioural rules. This reactive character of the simulation can be programmed in the script language (scripted control) or directly given by an external controller. We call the groups of virtual agents who apply the behavioural rules <*autonomous groups*>. Considering the levels of autonomy presented in this work, Figure 1 shows the priority criteria and Figure 2 shows an event treated by *ViCrowd*.

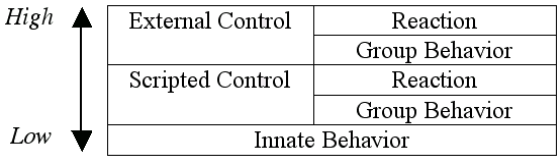

Figure 1: The behaviour priority

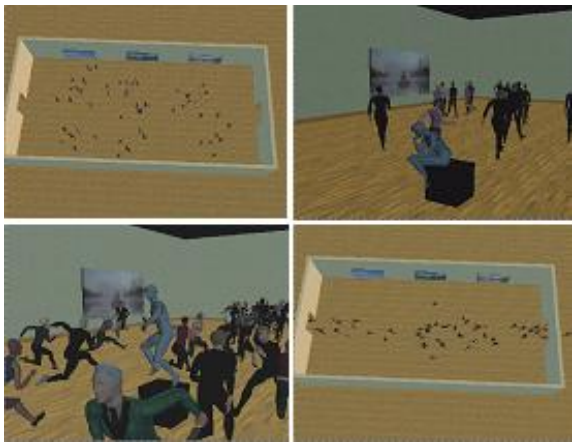

Figure 2: Scenes of simulation of evacuation due to a panic situation. Up-left and up right: before the panic situation, the crowd walks. Down-left and down right: crowd reacts because an event was generated when the statue becomes alive [Musse01].

3. Environment (*ENV*)

The *ENV* module is responsible for the management of information about the virtual environment (VE). This information can be the shortest path from a location to another, the location of obstacles and places, walkable regions, structured information inside the places (*e.g.* rooms), etc. The other modules that access *ENV* are *CROWD*, which needs

positions of places and paths to reach them, and *IC* which needs information of places and objects to trigger events. The VE handled by *ENV* is a geometric file where visibility graphs are built, and then *ENV* is able to compute shortest paths and inform them to the other clients. Figure 3 shows the *ENV* tool where the VE is edited.

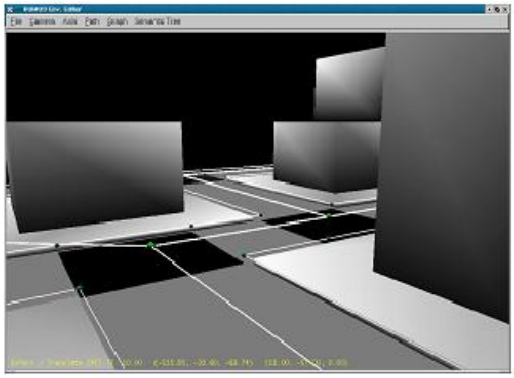

Figure 3: *ENV* tool

3.2 Main Module

The integration between the modules *IC*, *CROWD* and *ENV* is made using the Python language [Pytho00]. We have a main module in Python responsible for this integration. This module also sends geometrical information (position of agents and mobile objects) to the viewer module, as described in Section 4.

When one module needs some information from other modules, the *Main Module* is responsible for searching for this information, working like a server as shown in Figure 4.

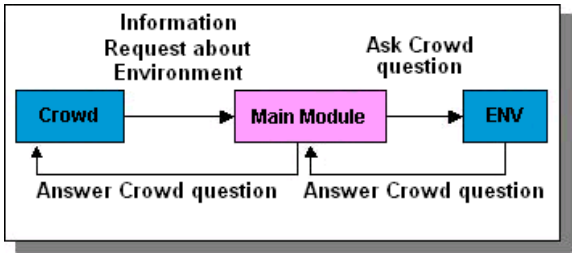

Figure 4: Main Module integration

We use a client-server architecture, where *Main* is the server, and other modules are clients. The *Main* module is a different server, because it decides which methods will be called to respond to a specific query. For the moment, we have only LOCATIONAL queries, which requires location data, for example: The *CROWD* module asks to *Main* what is the restaurant's position in the environment. *Main* sends the query to the *ENV* module. In this case, *Main* translates the query and decides which method of *ENV* is more appropriate to respond this query. Then, it calls this method and send the result of this query to *CROWD*.

The decisions taken by the server to select which module can answer a given query are based on a set of rules. These rules define how the interaction between the clients will be managed by the server. We call this server an "*active server*", because it assumes some autonomous behaviours according to the rules. Figure 5 shows rules that specify which modules should answer LOCATIONAL queries. For instance, the first rule describes that queries for a specific `PLACE` have to be directed to the *ENV* module.

```
SEND LOCATIONAL(PLACE) QUERY TO ENV
SEND LOCATIONAL(AGENT) QUERY TO CROWD
SEND LOCATIONAL(GROUP) QUERY TO CROWD
```

Figure 5: Rules for the "active-server"

3.3 Architecture

All modules are defined independently of each other using the object-oriented paradigm. We have public methods for communication between *Main* and the other modules. These methods depend on module functionality. For example, if a module needs a place position, then it should have one method to request this information. Each module should contain, at least, the following methods:

- `Init` (It initializes the module. It also sets a status flag signaling that the module is being used)

- `GetStatus` (It returns the status flag that signals whether the module is being used or not)
- `Query` (It returns a list of queries to be solved by the server)
- `Answer` (It is used by the server to respond to the module's queries)
- `Perform` (It calls the main method of the module)

The `Init` method is called by the user to initialize the module and to inform to the *Main Module* that a specific client will be used.

For example:

```
IC.Init()                                (1)
```

The line (1) above initializes the *IC* module and sets a flag to indicate that this module is being used. `GetStatus` and `Perform` methods are used by the *Main Module*. The first one is used to verify whether the module was loaded by the user. The second method starts the main method of the module.

Moreover, we have a script language in Python for the users to define the simulation. The user specifies all information about the simulation and starts it. Consider, for example the script:

```
Import Env                          (1)
Import IC                           (1)
Import Crowd                        (1)
Import Main                         (1)

IC.Init()                           (2)
Crowd.Init()                        (2)

IC.LoadFile("Camera.cfg")           (3)
Env.LoadFile("Environment.dat")     (4)
Crowd.CreateCrowd("Crowd.cfg")      (5)
Main.Run()                          (6)
```

The user imports the modules *ENV*, *IC*, *CROWD* and *Main* (1). Then *IC* and *CROWD* are initialized (2). `IC.LoadFile` method loads all definition of intelligent camera: events and entities to search during the simulation (3). Afterwards, `Env.LoadFile` loads all information about the environment described in `Environment.dat` file (4). `Crowd.CreateCrowd` method creates a crowd for the simulation from a configuration file (5). Finally, `Main.Run` starts the simulation (6).

Another important fact is the dependency between modules. This dependency is shown in Figure 6. The module *IC* depends on the modules *ENV* and *CROWD*, because these modules have the information that the intelligent camera needs to trigger events and position itself. The *CROWD* module depends on the *ENV* module, because it needs the environmental data that is owned by *ENV*.

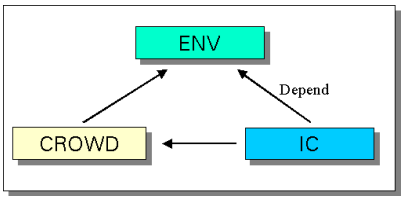

Figure 6: Dependencies between modules

When the user calls the `Main.Run` function, the *Main Module* starts the simulation. For every loop of simulation, the *Main Module* treats the queries, calls the `Perform` methods and sends geometric information about the agents and camera position to the viewer module. Then, the viewers display the result of the simulation (see Section 4). Figure 7 shows the overview of our framework.

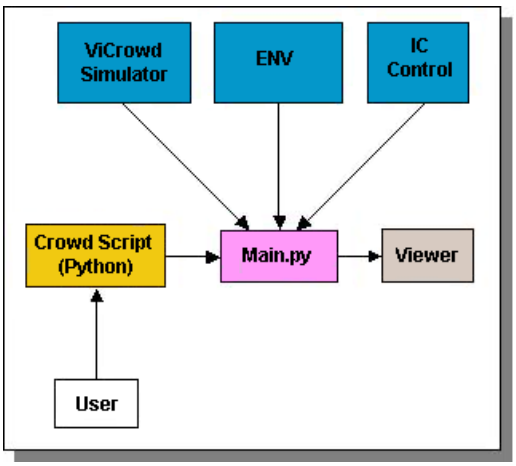

Figure 7: System Architecture

3.4 Case-study

To see how this system works in practice, we will study a sample simulation: the context is a panic situation. We have people walking on the street when a panic event is triggered. We use the *IC*, *ENV* and *CROWD* modules for this experiment. We define one event for *IC* to search during the simulation: a panic event. When this happens the camera shows people who are running. The *CROWD* module asks the *Main Module* where is the region to escape and then people start to go there. The *Main Module* sends *CROWD* queries to the *ENV* module and then returns the answer to *CROWD*. The *CROWD* simulator generates all crowd steps and send this to the viewer module. This simulation can be visualized in real time. In Figure 8 we can see an image of the simulation including 15 agents.

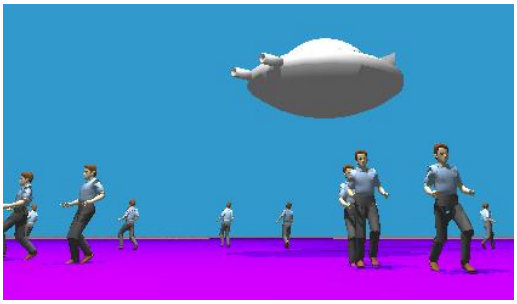

Figure 8: Panic simulation

We have two configuration files and one environment database: `Crowd.cfg`, `Camera.cfg` and `Environment.dat`. The `Camera.cfg` and `Crowd.cfg` files are described in Figures 9 and 10.

In the sample code we define one group, and their reactions when the panic event is triggered. And finally, `Environment.dat` is shown in Figure 11. It defines the obstacles (OBJ) and the exit positions (space position and rotation).

```
CROWD_SCRIPT
...
GR_0 GROUP_NATURE AUTONOMOUS
NB_AGENTS 13
PATH_DEFINITION FROM RESTAURANT
                TO PARK
GROUP_BEHAVIOR  2 ADAPTABILITY
          COLLISION_AVOIDANCE
END_GROUP
...
panic WHO GROUPS 1 GR_0
  WHEN FRAMES 310 1000

reaction1 panic
  GOTO 2 ESCAPE_AREAS
  CHANGE_EMOTION FEAR
end_reaction
```

Figure 9: Crowd configuration file

```
IC EVENT: WHEN GR_0 NEAR ESCAPE_AREAS
```

Figure 10: Camera configuration file

4 VISUALIZATION

In our framework, visualization is completely isolated from the behavioural model in order to work with any module. In fact, visualization is accomplished by a separate process that runs concurrently with the process responsible for the behaviour.

The separation of the behaviour and visualization modules has two major benefits. First, it keeps the system components loosely coupled, which allows the development of both modules independently. With this separation, each module can be developed by an expert in its area. This helps to create a system that is equally good in artificial intelligence and computer graphics.

The second advantage is flexibility. The framework is not limited to a single viewer or a single behavioural model. In other words, it is possible to use the framework with a number of different viewers and behavioural models, where the only constraint is to generate rules to create the relationship between the modules. It is also important to emphasize that, since all the modules follow the same interface, it is possible to combine every viewer module with every behaviour module with minimal work. This is desirable because different viewers may be adequate for different situations or the same viewer may be adequate for several behavioural models.

```
RESTAURANT
 POS ( -1609.88 0.00 0.66 )
 ROT ( 0.87 0.00 -0.48 )

PARK
 POS ( -727.00 0.00 -284.87 )
 ROT ( 0.68 0.00 0.72 )

OBJ_0
    BOUDING_BOX <Dimension>
    LIST_OF_VERTEX <List>
    LIST_OF_FACES <List>
END_OBJECTS_INFO
```

Figure 11: Environment file

4.1 Integration Between Processes

In order to visualize the simulation, the viewer process must receive information from the simulator process. Hence, there must be a communication channel between them. This can be seen as a simple producer/consumer problem, in which the simulator produces data that is consumed by the viewer.

In our implementation, the communication between the processes uses shared memory. On each simulation step, the simulator writes to the shared memory all the relevant data about the simulated entities (this can include, for example, the position, orientation and type of every entity).

Synchronization issues must also be considered. It must be ensured that the viewer displays every frame generated by the simulator, *i.e.*, the viewer must update its display for every simulation step. Furthermore, we must certify that the data on the shared memory is in a consistent state.

We are using a mutex-based scheme for synchronization. Whenever a process needs to read from or write to the shared memory, it must lock it. While the shared memory is locked, the other processes are not allowed to access it. After the data is read or written, the shared memory must be unlocked, in order to allow its access by other processes.

4.2 Viewers

At this time we have two viewers developed for the framework. The first one is a simple OpenGL-based [Woo96] viewer called *Caterva*. The models (for agents and scenario) are read from files in Wavefront's OBJ format [Wavef91].

Caterva is designed to be capable of displaying a large number of agents. The 3D models it uses are very simple, using few polygons. It is limited to receive and display agents' position, orientation and colour. Caterva is shown in Figure 12.

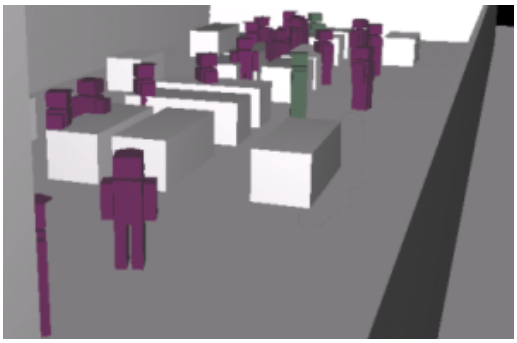

Figure 12: The *Caterva* viewer

Our second viewer is named *RTKrowd*. It is based in the *RTK Motion* toolkit, a commercial product developed by Softimage [Softi01].

RTKrowd is a more realistic viewer, capable of displaying more complex models, with more polygons and textures. Also, it is not limited to display only the agents' position and orientation: *RTKrowd*'s actors are capable to perform key-framed animations. The 3D models it uses are read from files in the Softimage's XSI format, which includes both geometry and animation. *RTKrowd* can be seen in Figure 13.

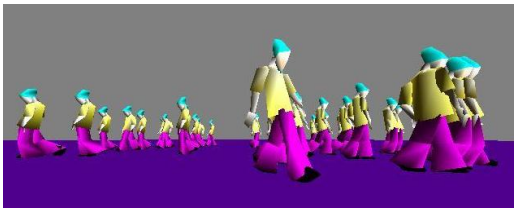

Figure 13: The *RTKrowd* viewer

5 CONCLUSION

We have described in this paper a framework which aims to integrate different behavioural modules in a client-server architecture. A novel idea of this paper is to present the possibility of integrating different modules using an "active server" based on a rule based interface. In this way, the server behaves in an autonomous way in order to treat the dependencies between the plugged modules. From the modules point-of-view the only constraint is that the modules have to provide a pre-specified number of functions to establish the minimum communication with the server. To illustrate the framework concepts, we discussed an experiment involving *IC*, *CROWD* and *ENV* modules to simulate a panic situation.

Another contribution is the possibility to integrate any viewer with different levels of details, as shown with the integration with *Caterva* and *RTKrowd*. We have chosen to develop two viewers with opposed goals: speed × visual quality. We are currently developing a third viewer module, that will be used to generate photo-realistic videos from our simulations. It will not be able to display the simulation in real-time as the other viewers, but the framework is flexible enough to support it.

As future work, we expect to improve the system's scalability by optimizing the communication between the modules. A possible approach is to add some memory to the *Main* module. This enables some enhancements, for example: when the *Main* module needs information from the *Crowd* module, it could ask only for the data that changed since the last query. An alternative approach is to save frequently needed data in a memory area accessible to all modules.

REFERENCES

[Aylet01] Aylett,R., Cavazza,M: Intelligent Virtual Environments–A State-of-the-art Report, *State of art reports, Eurographics*, 2001.

[Bicho01] Bicho,A., Raposo,A., Magalhães,L: Control of Articulated Figures Animations Using Petri Nets, *IEEE Proceedings of Sibgrapi*, pp. 200–207, 2001.

[Blumb95] Blumberg,B., Galyean,T: Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments, *Proc. SIGGRAPH, Computer Graphics*, pp. 47-54, 1995.

[Kallm00] Kallmann,M., Monzani,J.-S., Caicedo,A., Thalmann,D: A Common Environment for Simulating Virtual Human Agents in Real Time, *The Fourth International Conference on Autonomous Agents, Workshop 7: Achieving Human-Like Behavior in Interactive Animated Agents*, pp. 55–59, 2000.

[Matar94] Mataric,M.J: Learning to Behave Socially, in *D.Cliff, P.Husbands, J.-A.Meyer and S.Wilson, eds, From Animals to Animats: International Conference on Simulation of Adaptive Behavior*, pp. 453–462, 1994.

[Musse98] Musse,S.R., Babski,C., Capin,T., Thalmann,D: Crowd Modelling in Collaborative Virtual Environments, *ACM VRST/98*, 1998.

[Musse01] Musse,S.R., Thalmann,D: Hierarchical Model for Real Time Simulation of Virtual Human Crowds, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 7, No. 2, pp. 152–164, 2001.

[Noser96] Noser,H., Thalmann,D: The Animation of Autonomous Actors Based on Production Rules, *Proc. Computer Animation*, 1996.

[Perli96] Perlin,K., Goldberg,A: Improv: A System for Scripting Interactive Actors in Virtual Worlds, *Proc. SIGGRAPH, Computer Graphics*, pp. 205–216, 1996.

[Pytho00] Python Language Website. http://www.python.org, 2001.

[Reyno87] Reynolds,C: Flocks, Herds and Schools: A Distributed Behavioral Model, *Proc. SIGGRAPH, Computer Graphics*, Vol. 21, No. 4, 1987.

[Reyno99] Reynolds,C: Steering Behaviors for Autonomous Characters, *Game Developers Conference*, 1999.

[Sanza01] Sanza,C., Heguy,O., Duthen,Y: Evolution and Cooperation of Virtual Entities with Classifier Systems, *Workshop on Computer Animation and Simulation, Eurographics*, pp. 171–194, 2001.

[Softi01] Softimage Co. http://www.softimage.com, 2001.

[Treib99] Treiber,M., Hennecke,A., Helbing,D: Microscopic Simulation of Congested Traffic, in *Traffic and Granular Flow'99: Social Traffic, and Granular Dynamics*, 1999.

[Tu94] Tu,X., Terzopoulos,D: Artificial Fishes: Physics, Locomotion, Perception, Behavior, *Proc. SIGGRAPH, Computer Graphics*, 1994.

[Wavef91] Wavefront Technologies: *The Advanced Visualizer–User's Guide*, Appendix B, 1991.

[Woo96] Woo,M: *OpenGL Programming Guide: the official guide to learning OpenGL*, version 1.1., 2^{nd} Edition, Addison Wesley, 1996.