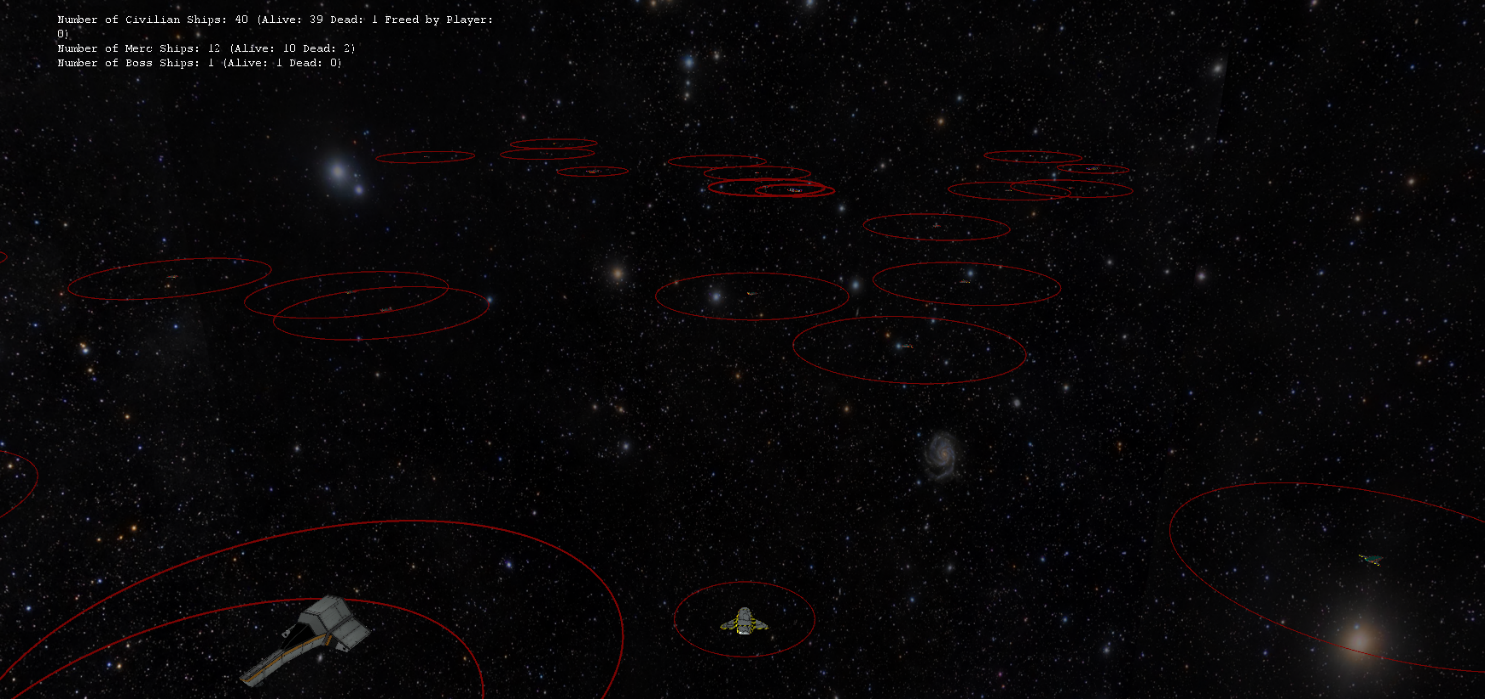
IS71027A: AI for Games

**Visualisation of steering behaviours with reactive agents**

**Ryan Singh**

**27/03/2015**



# Overview

The project is a 3D visualisation to begin creating a framework that provides steering behaviours for agents and providing reactive agents that react to each other concurrently throughout the simulation. The reactive agents are effectively Finite State Machines which use steering behaviours to move around in the visualisation. The idea behind the project was to learn the fundamentals of State-Driven Agent Design and steering behaviours and understand how it works mathematically.

The aim of the project was to become a framework for future work with AI behaviours. It was written in C++ and OpenGL, using Andy Thomason’s Octet Framework and Bullet Physics.

Contents

[Overview 2](#_Toc415253981)

[Chapter 1: AI Techniques Used 4](#_Toc415253982)

[Finite State Machines 4](#_Toc415253983)

[Autonomous Agents 5](#_Toc415253984)

[Steering Behaviours 6](#_Toc415253985)

[Seek/Flee 6](#_Toc415253986)

# Chapter 1: AI Techniques

## Finite State Machines

One of the core parts of this project is the implementation of State-Driven Agent Design via the use of finite state machines, or FSMs as they are commonly referred to. Finite state machines is a very old but popular tool to ‘*imbue a game agent with the illusion of intelligence*’[[1]](#footnote-1).

The origins of a finite state machine lie with mathematicians, using rigidly formalized devices to solve problems. *Programming Game AI by Example* authored by Mat Buckland specifies that;

*“A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.”*

For developing games, the main notion of using a FSM is to decompose an objects behaviour into manageable states. One of the earliest examples of this being used in games are the behaviours of the ghosts in Pacman. Famously, the ghosts will find Pacman while traversing the maze and “seek” him but if he eats a power pill, they will perform an opposing action and “flee” Pacman. Therefore, the input of the player eating one of the power pills is the condition for changing states. See figure 1.1.

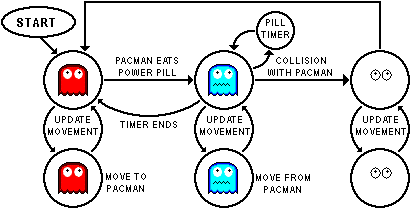


Figure 1.1. The ghosts are basic FSMs with different states depending on their environment and the state of Pacman.

Finite state machines are extremely useful tools because they can be visualised very easily, allowing developers to see the big picture, tweaking and optimizing the results. See Table 1.1.

|  |  |  |
| --- | --- | --- |
| Current State | Condition To Transition | State Transition |
| Fleeing | Safe | Wander |
| Wander | Spotting Enemy | Seeking |
| Wander | Spotting Friendly | Flocking |
| Seeking | Threatened | Fleeing |

Table 1.1. A simple state transition table (currently used in the project).

Complex FSMs can be achieved using a stack-based FSM, which ensures that the execution flow is concise without negatively impacting the code.

## Autonomous Agents

An Autonomous Agent is a term often used to describe an agent that possess a degree of autonomous movement. Often, an autonomous agent is an agent that has the ability to solve problems such as stumbling upon an unexpected situation such as being threatened and having the ability to respond. *Russell & Norvig (2003)* classify agents into five classes depending on their degree of perceived intelligence and capability:

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents
5. Learning agents

These agents vary in functionality and the latter become more capable of mimicking an intelligent system. The simple reflex agents’ functionality is based on the *condition-action rule*: if condition then action. Whereas the learning agents can potentially operate in a completely alien environment and become more competent at its tasks than its initial knowledge alone might allow. Learning agents are widely used in many different branches of AI from FSMs to Artificial Neural Networks or ANNs.

For the project, the group of techniques for action selection known as reactive planning are used by the autonomous agents (the different ships). The agents use the *condition-action rule* and such rules are often called productions. For the ships, the production rules are organised in a relatively flat structure where the production rules are not organised in a hierarchy. An extension of the project would be the implementation of a hierarchy, allowing layers to be organised in a simple stack, with higher layers subsuming the goals of the lower ones. This addition could provide a nice blend of steering behaviours.

The movement of an autonomous agent can be broken down into three layers:

**Action Selection**: Responsible for choosing its goals and deciding what plan to follow. This is the part that is often comprised of a FSMs which tell it what to do, when and how.

**Steering**: This part is responsible for calculating the desired trajectories to follow out the commands by the action selection (FSMs). This layer is where steering behaviours are implemented.

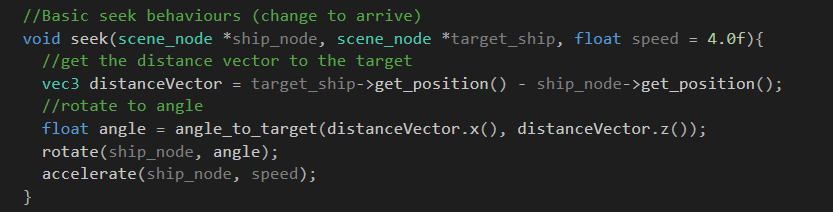
**Locomotion**: This layer represents the mechanical aspects of an agents movements. It essentially comprises of *how* to move from A to B by using a certain steering behaviour with different types of locomotion.

## Steering Behaviours

Steering behaviours aim to help an autonomous agent move in a realistic manner using particular algorithms such as Craig Reynolds’ wandering behaviour. Most steering behaviours are not based on complex strategies involving advanced path planning or environment awareness, but instead use local information, such as neighbours’ forces. Most of the steering behaviours were proposed by Craig W. Reynolds. The implementation of all forces involved in these steering behaviours can be achieved using math vectors.

### Seek/Flee

A seek steering behaviour returns a force that directs an agent toward a target position. To work out the desired velocity, you need to subtract the target vector by the current vector. This is the code used in the project to work it out;



To work out the angle or the shortest angle to rotate a model by, you mind find the arc tangent of two functions. By performing atan2(distanceVector.x(), distanceVector.z()), you are given the shortest angle from the current position vector to the target vector. See Figure 1.2.

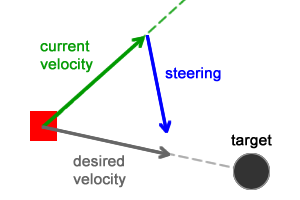
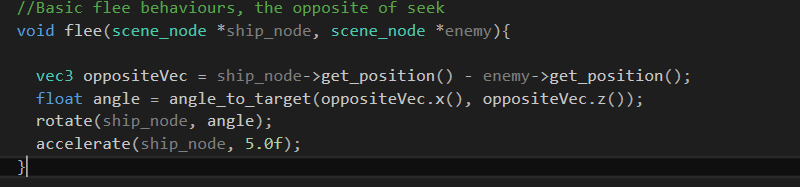


Figure 1.2: Calculating the appropriate vectors for the seek behaviour. The desired velocity is the vector of the resulting subtraction between the current velocity vector and the target vector.

The Flee steering behaviour is the exact opposite of Seek. Where seek gets the desired velocity to the target, flee gets the desired velocity that is the inverse of the target, creating a force that steers the agent away.



### Arrive

The seek behaviour is very useful for getting an agent to a target position but it will suffer from approaching the target velocity at a constant speed, often overshooting the target and infinitely trying to reach the goal. To combat this, Arrive steers the agent in such a way it decelerates onto the target position. Often, when an agent enters a ‘slowing area’, its velocity is ramped down linearly to zero. This is achieved by adding a new steering force relative to the agent’s velocity vector and dependant on how close they are to the target.



Once the agent is in the slowing area, the length of the distance vector can be used to define how the velocity is linearly ramped down. For the project, I have mapped it so that the float distance will be between 0.0f and 20.0f and the acceleration will be between 0.0f and 7.0f. Therefore when the distance is at its farthest, the acceleration is 7.0f and when it has reached its target, it would have been ramped down to an acceleration force of 0.0f and it will stop. See Figure 1.3.

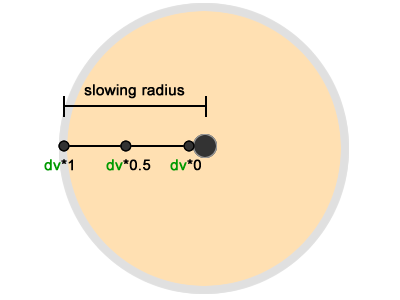


Figure 1.3. An example of an agent entering a slowing circle towards the target position.

### Wandering

Wandering is a very useful tool for creating agent behaviour. The idea behind it is to create a force that will give an impression of a random walk through the agent’s environment. There are many ways to create the illusion of random agent wandering but one of the most basic and impressive solutions is Reynolds’ of circle projection. The solution projects a circle in front of the agent and a follow a target that is constrained to move along the perimeter of the circle. The target is given a random displacement along the perimeter of the circle every step so that the agent appears to be alternating around jitter-free. See Figure 1.4.

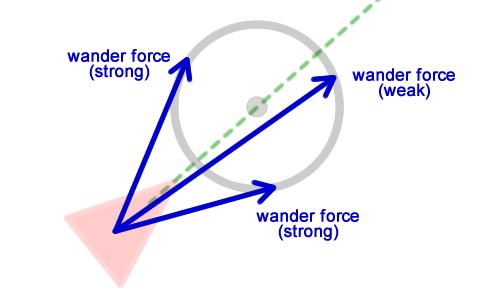


Figure 1.4. An example of how this solution might be visualised. The circle is displaced in front of the agent.

# Chapter 2: Implementation

## Autonomous Agents

For the project, I decided to have different steering behaviours depending on the state of the agents. The demo has 3 different agent types and a player. For naming conventions, the three different types of agents are; Civilians, Mercenaries, Bosses.

Each of the agents are updated every frame and their states are constantly checked and can change dynamically depending on production rules of each agent. Each agent reacts in a particular way with the other agents within the environment and they perform basic steering behaviours in accordance to the state. See table 2.1, 2.2 and 2.3 for the transition table for the civilians, mercenaries and boss respectively.

|  |  |  |
| --- | --- | --- |
| Current State | Condition To Transition | State Transition |
| Wander | Spotted Mercenary | Flee |
| Wander | Spotting Boss | Flee (slower) |
| Wander | Spotted Player | Arrive/Flock |
| Arrive/Flock | Player out of range | Wander |
| Flee | Safe | Wander |

Table 2.1. State transition table for the civilians.

|  |  |  |
| --- | --- | --- |
| Current State | Condition To Transition | State Transition |
| Fleeing | Safe | Wander |
| Wander | Spotted Civilian | Attack/Destroy |
| Attack/Destroy | Dead Civilian | Wander |
| Wander | Spotted Boss | Fleeing |

Table 2.2. State transition table for the mercenaries.

|  |  |  |
| --- | --- | --- |
| Current State | Condition To Transition | State Transition |
| Wander | Spotted Mercenary | Attack/Destroy (80% chance) |
| Wander | Spotted Civilian | Attack/Destroy(60% chance) |
| Attack/Destroy | Dead Mercenary | Wander |
| Attack/Destroy | Dead Civilian | Wander |

Table 2.3. State transition table for the bosses.

Each agent has a circle drawn around them to show their reactive radius’. If another agent goes within their radius then they will react accordingly to the agent. For example, a boss radius is larger than a mercenary, therefore, may find a mercenary before the mercenary can flee because his radius did not pick up the boss near him. See Figure 2.1.

References:

* [Russell, Stuart J.](http://en.wikipedia.org/wiki/Stuart_J._Russell); [Norvig, Peter](http://en.wikipedia.org/wiki/Peter_Norvig) (2003), [*Artificial Intelligence: A Modern Approach*](http://aima.cs.berkeley.edu/) (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, [ISBN](http://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-13-790395-2](http://en.wikipedia.org/wiki/Special:BookSources/0-13-790395-2), chpt. 2

<http://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-wander--gamedev-1624>

<https://unisalesianogames.files.wordpress.com/2011/08/programming-game-ai-by-example-mat-buckland2.pdf>

http://www.red3d.com/cwr/steer/

1. https://unisalesianogames.files.wordpress.com/2011/08/programming-game-ai-by-example-mat-buckland2.pdf [↑](#footnote-ref-1)