

# Simulating Crowd Behaviour in Computer Games

Robert Berggren

Luleå tekniska universitet  
Högskoleingenjörsprogrammet  
Datorspelutveckling  
LTU Skellefteå

## **Simulating Crowd Behaviour in Computer Games**



## Preface

---

---

This paper documents the work I did on the research for the ForeignGround demonstrator. My role in the project was to help evaluate a set of engines and to give suggestions on how to implement the AI. It is my hope that this paper will be useful for those working on the actual implementation of the demonstrator, as well as for other people who are considering adding crowd or group behavior to their game project.

As this work was a part of a larger project a lot of ideas came up during meetings and a lot of joint decisions were taken as a result of discussions. A few members have directly affected the design of the AI in their work, others have done so indirectly by visiting those meetings and taking part in the project. I'd like to thank everyone who worked on the Foreign Ground project; without them, my own work would not have been possible.

## Abstract

---

There are a lot of studies on how to simulate crowd behaviour. In this paper I first outline some of the previous work on the subject and then attempt to apply this to a specific game project. A lot of the work consists of finding a balance between features wanted in the game, limitations set by the game engine and time and resource management. The work resulted in a design proposal for the game AI. However, more interesting than the result is the conclusions leading up to it. Most important being the fact that it's not intelligence that's important, it's the appearance of intelligence!

### **Sammanfattning** (Abstract in Swedish)

Det har gjorts många försök att simulera beteendet hos folksamlingar. I den här rapporten går jag först igenom en del av det tidigare arbete som har gjorts inom området och försöker sedan applicera deras upptäckter på ett specifikt spelprojekt. Mycket av arbetet blev att försöka finna en balans mellan features man vill ha i spelet, begränsningar inbyggda i spelmotorn och tids- och resursbegränsningar. Arbetet resulterade i ett antal olika design-förslag för spelets AI. Mest intressant för allmänheten är dock inte resultatet, utan det arbete och de slutsatser som ledde fram till det. När det gäller spel är det viktigaste inte, som många tror, att karaktärerna är smarta; det viktigaste är att dom *verkar* smarta!

## Contents

---

---

<b>1.INTRODUCTION.....</b>	<b>5</b>
1.1.ABBREVIATIONS AND TERMS.....	5
<b>2.RELATED WORK.....</b>	<b>6</b>
<b>3.METHODS.....</b>	<b>8</b>
3.1.THE GAME ENGINE.....	9
3.1.1.UnrealScript.....	9
3.1.2.State-Machine.....	9
3.1.3.Events.....	10
3.2.AI REQUIREMENTS/DESIRES.....	11
3.3.HOW DOES THIS COMPUTE?.....	11
3.3.1.Life-like behavior.....	11
3.3.2.Crowd Size.....	12
<b>4.RESULTS.....</b>	<b>13</b>
1.1.DESIGN PROPOSAL 1.....	13
1.1.1.Navigation.....	14
1.1.2.State machine.....	15
1.2.DESIGN PROPOSAL 2.....	15
1.3.DESIGN PROPOSAL 3.....	16
<b>5.DISCUSSION.....</b>	<b>18</b>
<b>6.REFERENCES.....</b>	<b>20</b>

## 1. Introduction

---

Foreign Ground is the prototype for a game featuring modern graphics and advanced artificial intelligence. The aim is to create non-player characters that can act together as a group and whose behavior feel realistic. Since it is only a prototype time and resources will be scarce. Instead of creating a game engine from scratch the game will be built on top of an existing one, thus a number of engines are being evaluated. Since the game will rely heavily on AI an important factor in the evaluation is what AI-modules exist and, in particular, how extendable the engine is.

This work will consist of participating in the selection of a game engine and, based on the limitations of the chosen engine and the demands of the project, giving a proposal on how to design the AI.

### 1.1. Abbreviations and Terms

AI – Artificial Intelligence

Agent<sup>★</sup> – An AI-controlled character in a simulation or game.

Avatar<sup>★</sup> – When talking about games this usually refers only to the player's representation in the game. In simulations however (where there is no player) this is synonymous to Agent.

Bot<sup>★</sup> – AI controlled character

FPS – First Person Shooter, game genre

fps – frames per second, picture update frequency

GPU – Graphics Processing Unit

LoD – Level of Detail, separating objects into different levels of detail for processing, based on the proximity to the camera or user. Objects with high LoD are given more processing power, while objects with low LoD commonly use simplified algorithms and calculations or is updated less frequently. LoD is commonly used for graphics and AI.

NPC<sup>★</sup> – Non-Player Character

Mod – Modification of an existing game. Mods can be both big and small – ranging from simply adding another weapon to making it look and feel like whole new game.

Total Conversion Mod – “A big Mod”, using an existing game engine but creating new graphics, gameplay etc.

UT2004 – Unreal Tournament 2004, an FPS computer game.

XML – Extensible Markup Language

*★ On the use of Agent, Avatar, Bot and NPC: These terms all have roughly the same meaning. There are preferences on which term to use in different situations but they vary, especially between the gaming and simulating communities. The bottom line is that (with the exception of Avatar) they all always refer to a character which is controlled by some manner of AI.*

## 2. Related Work

---

Since Foreign Ground was meant as a training tool emphasis was put on realism, not only in graphics but also in character behavior. More specifically, we wanted characters to walk the streets and possibly gather into a crowd or mob. In this section I will present some of the previous work and research done in this area.

One rather impressive simulation was done by Tecchia et al. Over a set of papers they describe how they managed to simulate a crowd of 10000 avatars walking in real-time. At first they focused on collision detection [4] and the avatars were represented only by red dots. In subsequent papers they expanded on this by giving the avatars more human looks [5], environment-influenced behavior [6] and even shadows [7]. In order to accomplish this they used a 3D-model to generate a sets of images taken from discrete directions. Instead of using the 3D-model in the simulation they used an image from the set (decided by the heading of the avatar and the angle of the camera). Thus the 10000 avatars only took 1 polygon each in real-time, no matter how detailed the original model was. The shadows were done in a similar manner and depends on using a static light source.

Marchal [9] focused more on crowd behavior and on auto-generating additional data such as sidewalks, goals and crossings based on a simple city map. He claims that there are three main behaviors observed on pedestrians; monitoring (assessing the behavior of other people to avoid collision), yielding (changing trajectory to avoid collision) and streaming (following people heading in the same direction). In addition to this, the majority of pedestrians walk two by two and less than half walk alone. Large groups tend to walk in smaller subgroups. Thus some form of group behavior is needed to create a realistic environment.

Musse et al. created a crowd simulation in which avatars of a similar state of mind would join together in groups. The most dominating avatar in a group became the leader and dictated the goals and interests for the rest of the group. An avatar could change group if it found one it was better emotionally aligned with. They had no specific flocking algorithm, but got this “for free” since all the avatars in a group had the same set of goals. As an added incentive the groups could be set to wait for all the members to reach a specific goal before heading on to the next one. In between goals the avatars moved as individuals with no attempt to keep the group together or to maintain formation patterns (such as walking side-by-side). This works acceptably if the goals are close enough to each other, but over longer distances the group risks being split up as avatars adjust their heading and speed in order to avoid collisions. Furthermore, the absence of formations seems to have made the avatars default to walking in a line.

A well-known behavioral model was invented by Reynolds [2] in an attempt to simulate the movement of flocks of birds. He called the avatars “boids” (for bird-oids), and I will use that term as well. In Reynolds model the boids have three steering behaviors on an individual level that leads to their flocking behavior as a group. They are:

1. Separation: avoid collisions with nearby flockmates
2. Alignment: attempt to match velocity with nearby flockmates
3. Cohesion: attempt to stay close to nearby flockmates

Each of these three behaviors gives raise to a vector, which are then combined into what becomes the speed and direction of the boid. With all the boids moving independently, yet taking the rest of the flock into consideration, it is indeed a great model for simulating animals moving as a flock (such as birds, sheep and fish). It does not, however, work very well for human groups and crowds.

Many years later Reynolds presented another paper [3] showing how the same model could be used for other types of behavior. While the original model combined *Separation*, *Alignment* and *Cohesion* to create the combined behavior *Flocking*, other simple behaviors such as *Wall Following*, *Path Following*, *Seek* and *Flee* could be combined to create *Leader Following*, *Crowd Path Following* and more.

### 3. Methods

---

The first half of the project was spent evaluating the game engines and determining more specifically what would be required of the game AI. Once the engine had been chosen the rest of the time was spent further exploring what possibilities and limitations this presented us with.

Working with an already complete engine has both pros and cons. It was important to find an engine that would require as little extending as possible, while still being modifiable. It needed good graphics by today's standards, but most important was the possibility of implementing the AI and functionality we wanted to show with the demonstrator.

The engines we evaluated were:

- CryEngine (FarCry)
- Source (Half Life)
- Battlefield 1942
- Doom 3
- UnrealEngine (Unreal Tournament 2004)
- Operation Flashpoint
- Full Spectrum Warrior

The engines were evaluated based on a list of about 15 criteria, including graphics, scripting, editor and modding community. Also, the pipeline and workflow were big factors. We narrowed it down to two candidates: CryEngine and UnrealEngine. These were further tested by doing a small AI-implementation (a group of avatars maintaining a formation around the player) in each of them. It was hard deciding between these two engines. CryEngine is a newer engine and thus support better graphics and some other features (such as XML-support) which could be useful for the project. It uses a rather high-level scripting language which could speed up the AI-implementation. UnrealEngine on the other hand uses a rather low-level scripting language which very much resembles C++ and Java, and is almost as flexible. It is older than CryEngine, but has a large modding community and a long list of successful mods, some of which are Total Conversion mods.

In the end I think this is what became the deciding factor; while CryEngine had some great features that would speed up development there was also the risk of running into problems at a later stage of development, such as not being able to implement some part of the AI with their scripting language. With UnrealEngine on the other hand we *knew* that we would be able to do everything we wanted, though we'd have to do more ourselves.

Please note that we make no claims that UnrealEngine is superior to the other engines evaluated, we simply deemed it the best option based on the needs of ForeignGround and this project.

### 3.1. The Game Engine

Once the engine has been decided upon we of course wanted to take advantage of as much complete features as possible, instead of writing them ourselves. This saves time, which is the whole point of using an existing engine in the first place. This will most certainly also have repercussions on how you design the rest of the game. For example, if there's a finished collision detection system that you can use for free, but that doesn't work with too small objects, you might opt to simply skip the collision on those butterflies you were planning on putting in the game, instead of rewriting the system. It all depends on the cost/importance relationship.

So what features of the engine were important to the AI?

- Pathfinding. Waypoint-based. Waypoints are placed by the level editor in UnrealEd. Can be extended with new subclasses such as FleeNode (a place the bots go to hide)
- Collision detection. Collision against bounding volumes. Cylindrical collision can be used for humanoid characters
- Not tile based
- 3D graphics
- Line-of-sight. SeeMonster() or SeePlayer() is called when another avatar is within the view field.
- Scripting language. Basically the whole game is written in UnrealScript. We are limited to the opportunities offered by this language. Writing our own PlugIns is not possible. No control over rendering pipeline or shaders.
- Finite State Machine. Used by the Unreal bots, very easy to write your own states.
- Virtual machine. Simulated threading.
- Events. Special functions called by the game engine.

#### 3.1.1. UnrealScript

Unreal Engine uses an object-oriented scripting language called UnrealScript. It is flexible enough to do just about anything in regards to AI logic. The syntax is very similar to C++ and Java, which makes it easy to pick up for most programmers.

#### 3.1.2. State-Machine

The bots use a state-machine which is especially easy to use because UnrealEngine simulates threading. Normally it is troublesome to write states that take time (counted in seconds rather than milliseconds) to complete, since you can't lock up the computer during that time. For example, a PickupItem-state. This might include walking to the item, running an animation and adding the item to your inventory. This would usually result in cumbersome solutions such as *solution 1*.

While *solution 2* looks more elegant on paper, it introduces the concept of help-states such as Moving and Animate. This can make the state-machine a lot more complex, since it somehow needs to remember to go back to the PickupItem-state after completing the Moving-state. This can be done in several ways, such as keeping a LastState-variable (but what if Moving in turn needs another help-state?), using a ChooseNextState-function (which does not guarantee that it will return to the last state, and risks making the avatars look stupid) or creating a different help-state for each state (PickUpMoving, AttackMoving, OpenDoorMoving etc.) and hardcoding which state to return to (which lowers re-usability and calls for a lot of excess code).

```
state PickUpItem
  • if at item
    • if animating
      • if animation done
        • go to new state
    • else
      • add item to inventory
      • run pick up-animation
  • else if moving
    • move slightly towards item (happens
      each frame until item is reached)
  • else start moving
```

*Solution 1*

```
state PickUpItem
  • if at item
    • add item to inventory
    • go to state: Animate(pick up)
  • else
    • go to state: Moving(item position)
```

*Solution 2*

The threading in UnrealEngine does away with this problem since it enables you to use functions that span over several seconds. With this our state might look like *solution 3*.

```
state PickUpItem
  • MoveTo(item position)
  • RunAnimation(pick up)
  • add item to inventory
  • go to new state
```

*Solution 3*

In this case `MoveTo()` will move the avatar little by little each frame, and won't return until the item has been reached.

### 3.1.3.Events

These are functions that are called when certain events occur. `SeeMonster()` and `SeePlayer()` are examples of this, and also collisions. Events can be used to change states. For example, an UT2004-bot might go to the Attack-state when it sees the player.

### 3.2. AI Requirements/Desires

ForeignGround is to be used to prepare soldiers who are going on peace-creating missions abroad. By playing through typical scenarios they will get a better idea of what to expect and how to react. Just as in real life violence will be possible and maybe even necessary in some scenarios. However, the player should always strive for the most peaceful solution possible and should follow the rules of conduct. Emphasis will be put on realism, cultural differences and resolving situations with communication. From discussions during the project meetings I came to some conclusions as to what was desired of the NPCs in the game.

- Life-like, culture-influenced behavior
- Should react to both player behavior and dialogue input
- Crowds of up to 200 people
- Fast and cheap to create (project development phase about 4 months)

### 3.3. How does this compute?

Our AI will be state-based, since state-machines are a common and proven way of implementing AI, and in this particular case was so easy to use and extend. The engine comes with *a lot* of code for behaviors and states for the UT2004 bots, teams and different gamemodes. Unfortunately a great deal of this is very game specific. Rather than going through all this code and disabling what we didn't want/need (we don't want our native Liberians double-jumping onto the rooftops) it was deemed easier to write the behaviors from scratch.

Some things we could keep were pathfinding, movement and collision detection. Basically, in terms of Reynolds [3] three layers, we could keep the *Locomotion Layer* but needed to implement our own *Steering* and *Action Selection* layers.

#### 3.3.1. Life-like behavior

One way to achieve this might be by using fuzzy logic in the Action Selection layer, and by each bot having thirst-, hunger- and other similar parameters that increase over time. This would result in very intelligent behavior. Unfortunately it would also take a long time to implement and would be difficult to debug. Marchal [9] said that “. . . goals are not to be taken exactly like real human goals such as shopping or picking up somebody at the station. Although the idea is to make the observer believe that agents are acting so, our goals turn to be some 'temporary places to go'.”

ForeignGround will be played in first-person view, or possibly with a camera slightly behind the main character. There will be no birds-view giving the player an overview of the area and the characters. Considering this, it is likely that, like Marchal, simply picking a waypoint at random will look intelligent enough. Realism can be enhanced by having the NPCs perform an action (such as speaking or playing an animation) when they reach a goal. The action can be dependent on what type of waypoint it is. In this way a character might walk to a random waypoint, find that it's a “ShopPoint” and run a “shopping”-animation. In the game this appears about the same as needing food, deciding to go shopping and finding a store, but is considerably easier to code and debug. With no overview of the area it is likely that the player won't even notice anything odd unless he follows the same NPC around for some time.

Note that all this only applies to when the NPCs are “relaxed”. *Events* will be used to keep the characters reactive to occurrences around them, such as seeing the player and waving. As for the culturally correct behavior, it is more an issue of implementing the right behaviors and having the right sounds and animations than it is a design issue, so I will discuss it no further.

### 3.3.2. Crowd Size

One thing I needed to find out was how large crowds we would be able to have. I tried adding a large number of characters (c:a 150) to the game. When they were all in view the frame rate went below 10 fps. However, when looking away from them (i.e. when they were culled), the frame rate went up to 50 fps. This was in a very simple scene (a large room basically) with no graphic effects to suck up GPU-power. While it is *possible* that they have some sort of LoD that allows the AI to “cheat” a bit with calculations while they are not on screen, the fact that the engine is written for fast-paced first person shooters and multiplayer leads me to doubt this. Hence the conclusion I draw from this test is that the crowd size will be limited by how graphically detailed the characters are. With no access to the rendering pipeline there is no hope of making optimizations. Assuming our characters are about as detailed as the UT2004-bots, I'd say we should try to limit the number of characters on screen to about 50, and not above 70. That's not to say that we can't have more characters in a scenario, but they should be spread out and unlikely to gather in one spot.

## 4. Results

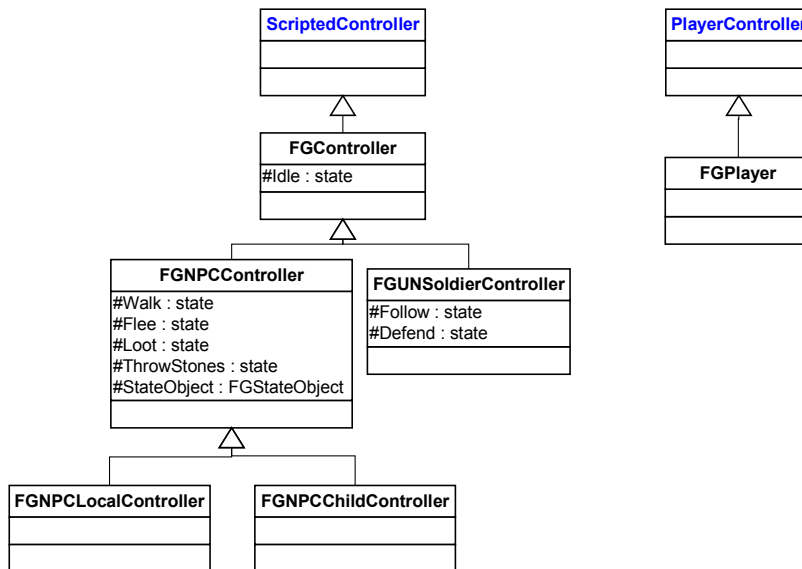
In this section I present three different design proposals for the Foreign Ground AI. The first proposal was done together with other members of the project. The other two are additional proposals that I designed based on my research on the subject of crowd behavior, the game engine and the project requirements. In general, the AI is rather simple in order to keep it easy to control and debug. It is state-based and states can be overridden in subclasses, which makes it easy to reuse and extend behaviors.

### 1.1.Design Proposal 1

A lot of the text and images in this section comes from a document written together with Fredrik Mäkeläinen. Refer to Appendix A for the full document.

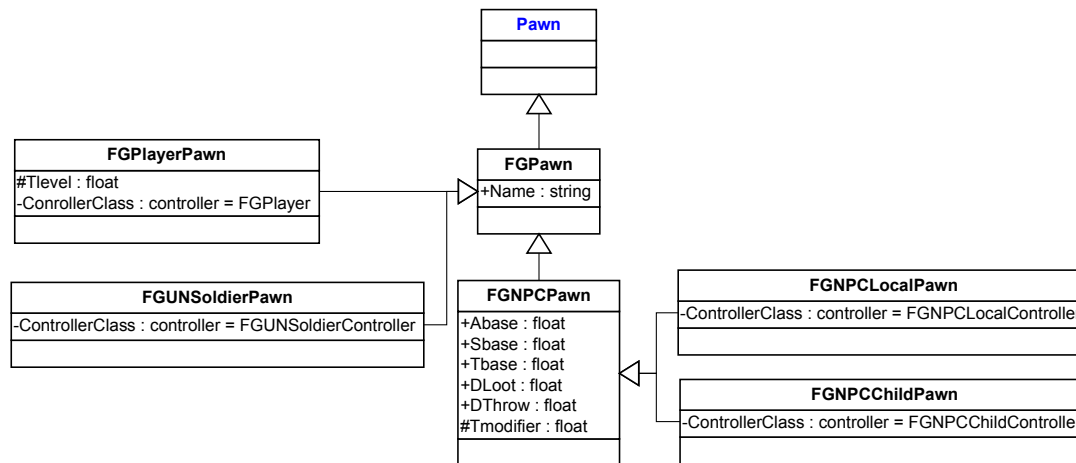
Class attribute prefix

- + attribute is editable from UnrealEd
- # attribute is hidden
- attribute is hidden and defined in a parent class



*Controllers* implements the states an agent can be in. States can be overridden in the subclasses.

A *Pawn* is the graphical representation of an agent. It can be placed in the editor and therefore also contains all parameters that is to be changeable from the editor. The pawn needs to be inherited to a new subclass each time you write a new controller, but you can have multiple pawn classes that use the same controller. This may be useful since you can set a default mesh on the pawn and thus facilitate the work of the level editor. (The level editor can place 5 child controllers instead of placing 5 NPC controllers and then changing the mesh and collision data for each of them.)



The three parameters *motivation* (*aggression*), *strength* and *threat* are used to make decisions. These parameters are calculated from base values that changes depending on the players actions in the game and modifiers that depend on the state of the game. Though they may change during implementation, here are the current formulas:

Motivation = Abase + *currDesire*( state )

Strength = Sbase + *friends*( radius )

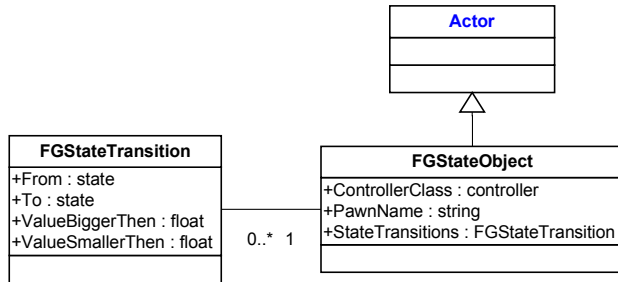
Threat = Tbase + *soldiers*( radius ) \* Tmodifier + *enemies*( radius )

The initial base parameters will have default values, but can be changed on an individual basis by the level editor in order to achieve different *initial* behaviors. The function *currDesire*( state ) will return a value describing how much the agent currently desires to do/achieve the given state. Exactly how the formula looks may be different for each state, but it will somehow incorporate a corresponding desire modifier (DLoot, DThrow etc.) that can be changed on an individual basis in the editor in order to achieve *permanent* different behaviors.

### 1.1.1. Navigation

We will use the built in pathfinding which requires the level designer to place NavigationPoints. Once placed UnrealEd can calculate the bindings automatically. Special points can be created to tell the bots where they can perform different actions. For example: LootPoint (a location where the loot state is available), FleePoint (while fleeing the NPCs will try to reach one of these), AttractorPoint (while wandering the NPCs will favour movement towards this location).

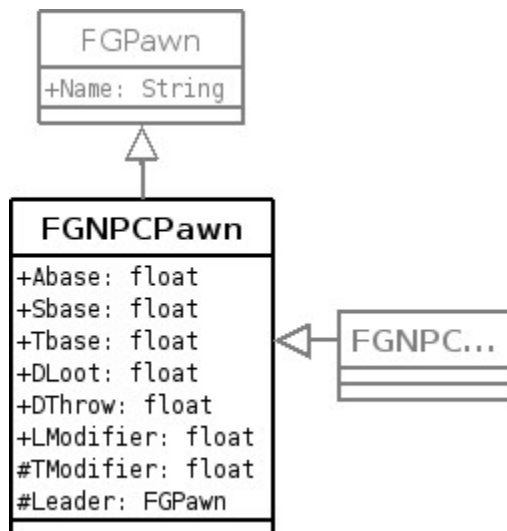
### 1.1.2. State machine



By placing one or more FGStateObject:s in the level the transition values for a controller (or even individual NPC:s) can be modified from within UnrealEd and no compilation or programmer intervention should be necessary. In effect, the level designer can modify the state-machine for a character, deciding what states will be available and what the [Motivation + Strength - Threat]-value must be in order to move from one state to another.

### 1.2. Design Proposal 2

With the first design the NPCs will gather at AttractorPoints until the group is big enough (the strength value is high enough) to have the confidence to decide on a course of action, such as looting or throwing rocks at the authorities. However, there is no group behavior to ensure that they decide on the same action, or location to perform it. Because they have different personalities it is entirely possible that they will gather in strength only to head of in different directions. Once separated they will loose confidence and head back to the AttractorPoint again. This all depends on the character personalities, what states (actions) are implemented and how scripted the scenario is. The risk is that it will result in some rather dumb AI. One easy way to prevent this without adding advanced group behavior is by adding the concept of a leader.



The only difference is the addition of two variables in the FGNPCPawn-class.

# Leader: The leader this NPC is currently following.

+ Lmodifier: Modifies how likely the NPC is to listen to its leader; how “headstrong” the NPC is. Editable from UnrealEd.

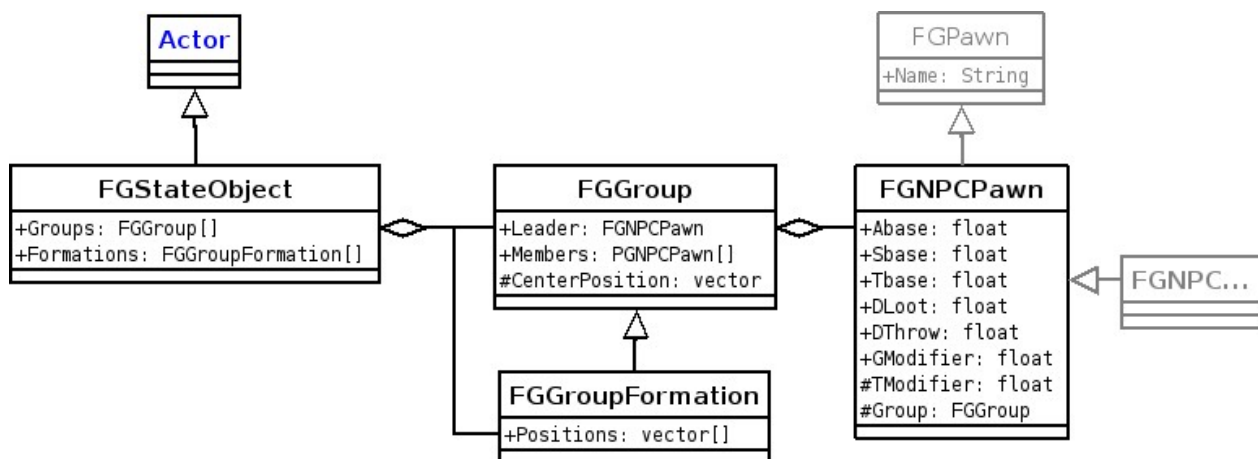
If two NPCs with the same desired state meet the one with the largest motivation will be chosen as leader. If they meet a third NPC with even higher motivation he will be chosen as the new leader. There will then be two formulas for calculating the motivation.

1.  $Abase + currDesire( Leader.currState ) + Leader.currDesire( Leader.currState ) * Lmodifier$
2.  $Abase + currDesire( state )$

Formula 1 will be used when calculating motivation for the state the leader currently desires. Formula 2 is used for any other state. This gives the NPCs an added incentive to stick to the same objective as the leader. Should they still choose a different objective it means that they want it badly enough to disregard their leader and head off on their own (the Leader-parameter is set to none).

### 1.3.Design Proposal 3

The second design allows for a simple form of group decisions. Each member of the group know who the leader is, but the leader does not know who is following him. Consequently, the members of the group can listen to their leader, but no “democratic” decisions can be taken. Furthermore, it's not possible to use any group movement algorithms, such as waiting for each other, or the behavioral model introduced by Reynolds [2,3]. With this in mind I created another design.



The Leader and LModifier-variables in FGNPCPawn has been replaced by Group and GModifier, respectively. FGStateObject contains lists of all the current groups. This may not be necessary but could probably come in handy at times. I mostly added it so you can place it in UnrealEd and predefine some groups.

FGGroup is a loose group, while the subclass FGGroupFormation has set positions for each member, relative to the center position of the group. It will need either a maximum number of members, or an algorithm for calculating positions. By subtracting the center position from the position of the leader, we get a directional vector that we will use to turn the formation.

One thing to watch out for is how the center position is decided, as it can severely affect the behaviour. For a loose group it could be either the “true” center position, that is the average position of all the members of the group, or it could be a point say 5 steps behind the leader. What you need to consider, when using the latter model, is how to update the center position. Will it be 5 steps behind the leader, and updated as soon as the leader turns? Will it only be updated when the leader moves? Will it drag behind, and only be updated when the leader moves away from it? This will affect any group but will be most obvious in a formation.

## 5. Discussion

---

Crowd Simulations usually try to simulate as large a crowd as possible. In order to achieve this they need to use simplified graphics and optimized collision and steering algorithms. Simulations such as these can be used for a variety of purposes, such as evaluating what effect a change in the road network in a city would have on traffic, before actually going through with construction.

In computer games the demands on the graphical quality are higher. This immediately rules out having huge crowds. Of course, this has the positive side-effect of making collision and steering less of an issue. The designs presented in this paper all draw on the theory that believable behaviour can be achieved simply by giving the characters random goals. This makes it appear as if they have things to do and places to go; they seem intelligent.

The first design is the most basic and only contains the essentials for the game AI. It still has some interesting features that allow the personalities and even the state machine to be edited down to an individual level, without recompiling. It makes no attempt at coordinated group behavior, relying instead on the level editor to set the characters' desires and state-machines in such a way that they will gather and perform the action desired in the scenario. This will create some rather static behaviour, as the characters will always choose the same actions up until the point where the player gets close enough to start influencing them. There will be no surprises such as "Oh, they're looting a truck this time!", or "Oh, this time they're raiding the shop!". To create different main problems for the player to solve the level editor must make a separate copy of the level and edit the characters' desires and/or state-machines.

This isn't necessarily a bad thing! It will require more work of the level editor but will be easier to debug, since you know exactly what behaviour you are trying to achieve. It will certainly lower replay value for someone who is playing the game just for fun. However, for educational purposes it may be good for at least the instructor to know what problems will arise in the game session, and to be able to choose a different scenario in the next session.

In any case, should the lack of group coordination turn out to be a problem the second design proposal can be implemented. It is actually just a small extension that aims at keeping a group focused on the same goal. With this you can add some randomness to the personalities/desires of the characters and they will still come together for a common goal. You will be able to create a scenario where they will loot either the truck or the store - or both! - and it will be different each time you play. This will of course set a higher replay-value but might be bad for educational purposes. However, "static" scenarios can still be created in the same way as before; by tweaking the state-machines and desires in the editor. In other words, there is no risk of losing usability.

Even with this small change you won't get any coordinated group behaviour. It merely assures that the characters in the group have the same goal in mind, not that they wait for each other, or stay in a formation, or try to coordinate an attack. If a wild mob is all that is needed, this will probably be enough. I suggest implementing the basic design and then adding the extension, if need be only (don't fix it if it ain't broken). If there are plans for a more “intelligent” group of people, it is my strong suggestion that a group-layer be added to the design from the start. While it would not take long to change the basic design to include a group class, it will affect a lot of behaviours and you risk having to re-write large portions of the AI. For example, adding Reynolds [2,3] flock movement would require us to completely re-write the movement layer in UnrealEngine, as each character would need to move according to a directional vector, updated each frame, instead of using the current MoveTo-function.

The designs presented in this paper are very specific for the UnrealEngine and the Foreign Ground demonstrator. What, you may be wondering, does this have to do with computer games in general? Well, while the resulting designs may not be of much interest, some of the decisions and conclusions leading up to them are. Before implementing a complex system that will make your characters “really smart”, think carefully:

- Do you really need this system? Perhaps there's a simpler way of achieving the same results.
- Will it look any different to the player? If it's not a significantly improvement over the simple solution, then why bother? Consider also the risk that the characters might end up looking dumber than before, because of the increased difficulty to debug and balance the system.
- Is there enough time? Resources are often very limited, and time spent on developing this system is time that could be spent on polishing other aspects of the game and AI.
- Even if there “should” be time, is it worth spending the resources on it? Weigh the system against other optional features you might have; which one would most benefit the game-play?

In conclusion: it's not intelligence that's important, it's the appearance of intelligence!

## 6. References

---

- [1] Dave C. Pottinger – Coordinated Unit Movement. Game Developer Magazine, January 1999. [http://www.gamasutra.com/features/game\\_design/19990122/movement\\_01.htm](http://www.gamasutra.com/features/game_design/19990122/movement_01.htm) {Acc. 2005-06-16}
- [2] Craig W. Reynolds – Flocks, Herds, and Schools: A Distributed Behavioral Model. SIGGRAPH '87, Computer Graphics 21(4), 25-34, July 1987
- [3] Craig W. Reynolds – Steering behaviours for autonomous characters. Game Developers Conference, Miller Freeman Game Group, 1999.
- [4] Franco Tecchia and Yiorgos Chrysanthou – [Real-time Visualisation of Densely Populated Urban Environments: a Simple and Fast Algorithm for Collision Detection](#). Eurographics UK 2000, Swansea, April 2000
- [5] Franco Tecchia and Yiorgos Chrysanthou – [Real-Time Rendering of Densely Populated Urban Environments](#). B. Peroche and H. Rushmeier, editors, Rendering Techniques 2000, 83-88, 2000. Springer Computer Science
- [6] Franco Tecchia, Celine Loscos, Ruth Conroy and Yiorgos Chrysanthou – [Agent Behaviour Simulator \(ABS\): A Platform for Urban Behaviour Development](#). Presented at the ACM/EG Games Technology Conference, January 2001
- [7] Celine Loscos, Franco Tecchia and Yiorgos Chrysanthou – [Real-time shadows for animated crowds in virtual cities](#). Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST) '01, 85-92, Banff, Alberta, Canada, November 2001.
- [8] Franco Tecchia, Celine Loscos and Yiorgos Chrysanthou. – [Visualizing Crowds in Real-Time](#). Computer Graphics forum, Volume 21, Number 4, December 2002, 753-765
- [9] David Marchal – [Simulating Pedestrian Crowd Behaviour in Virtual Cities](#). 2002, University College London: London
- [10] S. R. Musse and D. Thalmann – A Model of Human Crowd Behavior: Group Inter-Relationship and Collision Detection Analysis. Computer Animation and Simulation '97, 39-51, Eurographics workshop, Budapest, Springer Verlag, Wien, 1997