
Using AI with steering behaviors to model crowds

Project report as part of the COMP 791C PhD course
Concordia University
Nicolas Brodu, December 2004

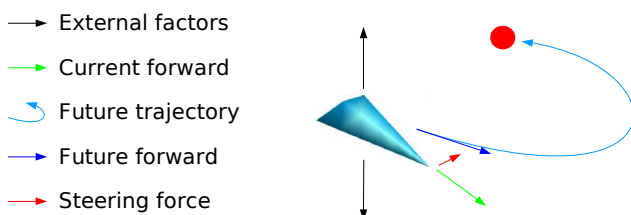
Motivation

In the 1999 Game Developers Conference, Craig Reynolds presented a way to model steering behaviors for autonomous characters¹. By using a point-mass model for agents and Newtonian dynamics, the agent motion is completely determined by the forces it is applied through time.

These forces fall into two categories:

- ◆ External environmental forces. Gravity immediately comes to mind, but isn't the only one in this category. The agent is usually passive with respect to these forces, and they are applied whatever the agent's decisions.
- ◆ Internal agent's decisions, based on its capabilities. In a car race game for example, the agent is limited by a maximum speed and acceleration. Within these limits, the agent is free to decide on which force to apply to achieve its goals. Similarly, all a player can do to act on the agent is choosing which forces to apply at which time (albeit through a friendly user interface, but the final result is the same as far as simulation is concerned).

Consequently, the goal of an AI is to produce these forces in such a way as to compete with the player. Schema 1 is an illustration of this: The AI of the agent has for goal to reach the given target. Given its current forward direction, the forces from the environment and its limitations, the AI chooses a steering force to apply. This gives a new forward direction, and implicitly defines a trajectory.



Schema 1: Steering force to apply to reach a target

The definition of steering forces to apply to several well-defined situations is given in¹. For example, steer for evasion of a mobile target, steer to seek for a fixed point, steer for exploring the environment in a random but consistent way,

¹ Steering Behaviors For Autonomous Characters, Craig Reynolds, 1999, Game Developers Conference, <http://www.red3d.com/cwr/papers/1999/gdc99steer.html>

etc. We can consider these as building blocks for the AI, and the role of the AI then becomes choosing and combining appropriate steering behaviors according to the intended goal.

This is the motivation that originally guided the project, and which in turn lead to the development of a complete environment. Indeed, for the AI to work on something, we first need to define a world, its physics, and a scenario. A case study is already available for steering behaviors in the form of the OpenSteer library², developed as concrete illustration of the aforementioned paper. One goal of this project is thus to extend the notions presented there, and provide a rich environment in which interesting challenges can be developed for the AI.

The world

The world developed in this project is a full 3D environment. To make this 3D aspect relevant agents are allowed to fly, and the terrain is not an infinite flat plane but rather a fully generated height field, with obstacles.

The world is either open in all directions, or made cyclic in both X and Y (cyclicality in Z isn't relevant in this case as we use a terrain). As simple as this may sound, it has numerous and not immediate technical impacts, including on neighborhood queries, terrain generation, and collision detection and avoidance. On the other hand, if the cyclic world is sufficiently large, this effectively suppresses boundary conditions, or the risk that agents disperse to infinity.

The following sections detail these points, and relate what difficulties were encountered in implementing them for this project.

Locality and neighborhood queries

The problem in finding neighbors is that the naive algorithm of running through all other agents and comparing their distances is inherently $O(n)$, which becomes $O(n^2)$ as we have to apply it on all agents each step. This is a strong limitation for scalability, and given the fact we also have to reserve some computational power for the AI itself, another method is necessary.

OpenSteer² uses a "SuperBrick" implementation. It consists in splitting the world along each dimension into a regular array of bricks. Then each block maintains a list of all the agents

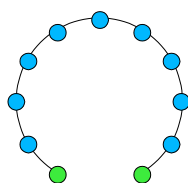
² <http://opensteer.sourceforge.net/>

located in its region. When an agent does a locality query, it can quickly eliminate all blocks farther than the query maximum distance, and run through the remaining block lists. This is a classical case of memory vs processing resource consumption, as a finer grid corresponds to a more precise elimination of far away blocks. In any case, when all agents are reasonably scattered around the world and they do only reasonably short distance queries, this algorithm provides a substantial improvement over the initial $O(n^2)$ one. This algorithm was thus reused in this project. It had to be modified to take into account world wrap, though. This isn't difficult, but isn't trivial either, and I made it so that the new cyclic query functions integrate with the original code in a fully compatible way.

Another algorithm can be used, which we'll call Neighborhood Propagation within this project framework. Though this algorithm was created independently for this project, it's so simple previous references to it can easily be found. For example in the Flies, and Mega Flies software by Keith Wiley³. It consists in each agent maintaining a list of its neighbors at a given time. Then, when an agent moves, it updates this list locally by checking on its previous neighbors, their own neighbors too, up to a maximum recursion level. Usually a simple second-order search is used, as otherwise the cost of this recursion would outweigh the benefits of its local nature.

One assumption in this algorithm is that agents don't move too fast, so that chances are an agent neighbors are still in the vicinity, or at least in its old neighbors vicinity. Unfortunately, this assumption has serious drawbacks, and fast moving agents aren't detected. Also, if an agent gets isolated from the group, it can never be found back. A way to overcome this second limitation would be to tweak the algorithm to maintain the last reference, even if it's too far away. But even this won't overcome the fundamental problem with this algorithm: two groups can cross without ever seeing each other (see schema 2).

The neighborhood propagation algorithm consists in maintaining a local neighbor list. It implicitly defines a metric: the recursion level necessary to find another agent. But it doesn't include a way to detect new neighbors based on their real distance. Consequently, two seemingly distant agents according to the algorithm metric may in fact be very close in distance, and completely miss each other.



Schema 2: Fundamental problem of neighborhood propagation

³ <http://www.unm.edu/~keithw/artificialLife/megaFlies.html>

In order to overcome this limitation, I implemented a mixed mode for this project. The agents use neighborhood propagation to quickly update their local list, and complement this by a locality query in the bricks database (before propagation for efficiency). The trick is to only include a fraction of the agents in the database, so that lists are much shorter in the bricks, and so that even in the case the agents are all located in a few bricks (or if the query distance is too large) the algorithm doesn't degrade to the $O(n^2)$ case. With this addition, the hope is that at least one agent in two meeting groups will be localized in the database, and included in at least one of the other group members neighbors. After a few updates, the whole groups become known to each other.

This mixed mode combines the best of both algorithms, without too much overhead. In addition, its reliability depends on only one parameter: the probability of an agent to be included in the database. By adapting this parameter to the situation, it's possible to trade neighbor detection reliability for speed, in a controlled way. This project currently does not update this probability at run-time. An interesting extension would be to devise an algorithm that does this automatically with clever heuristics, to allow speed increase in cases the situation is consistent enough with only neighbor propagation.

Finally, it's worth noting that obstacles on the terrain can help too. By making the obstacles (or more generally scenery items) participate in the neighbor propagation, the reliability of this algorithm is greatly increased, especially if the obstacles (more generally helpers) are uniformly spread all over the area. Moreover, by including the obstacles in the bricks database in the mixed mode, it is possible to ensure an agent-obstacle collision will always be detected.

Cyclicality

As previously mentioned, world cyclicality is not a big deal on a conceptual level, but its implementation is pervasive in the sense it has consequences on many other parts of the project. None of these consequences is in itself a great problem, and nearly all are straightforward to implement, but it's just that cyclicality has much more effects than I previously thought.

I already mentioned its effect of the locality queries. Another example: while it is easy to wrap a position, and with a little more care a position difference vector and a distance between two agents, these wraps should be included at all levels in the AI routines and steering forces instead of plain computations. It's even worse for agent to agent collision

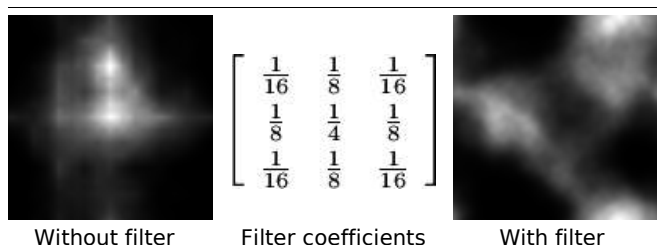
avoidance steering vectors: at least one ray has to be cast in the direction toward the estimated minimum distance future point along the two agent's trajectories, with the target agent translated to this point for ray intersection. Throw cyclic wraps in this, and that's more particular cases to handle.

Cyclicity also has an effect of terrain generation: the algorithm must take it in account and produce tileable maps for the height field. The normals must be averaged over both sides, and the world space vertex array must include one extra row in each dimension so the generated terrain can be placed side by side with itself (without this repetition, the triangles joining the last and first points would be missing).

Terrain Generation

The terrain generation technique I used for this project is inspired from Perlin Noise. The idea is to generate noise at different spacial frequencies, and apply interpolation filters to combine these different levels on the same scale. Instead of using the classical (bi/tri)linear filtering techniques, or spline-based techniques, I have used for this project a wavelet reconstruction. Indeed, wavelets are expressly designed for multi-scale frequency analysis, so it seemed a natural choice to replace the interpolation filter with a wavelet filter.

I have chosen the Daubechies W6 wavelet for this task. It is very simple to implement, and has a fractal nature, which I initially thought would be a nice property for a terrain. But whether this is an advantage or not given the smoothing I'll talk about below, is open to discussion. I guess other "natural" choices would be from the coiflet & spline families, or the bi-orthogonal Daubechies(9,7) wavelet which is used in the JPEG 2000 standard. Since none of these wavelet transforms are as simple to implement as the W6 filter, and since the potential benefits (or drawbacks) in using them is unclear for this project, sticking to W6 seemed a reasonable choice.

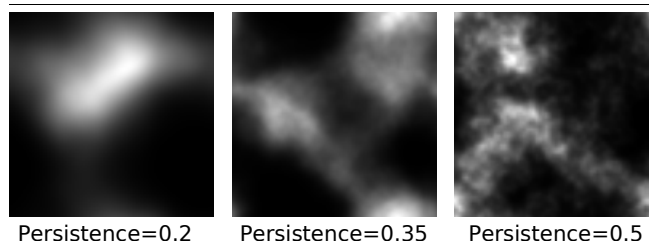


Schema 3: Necessity of a smoothing filter

What about the smoothing filter? A common problem when interpolating on both image dimensions independently is the apparition of blocky square artifacts. W6 is not exempt of this defect, though in this case the artifacts take the form of lines and square spatial echos (see

Schema). These may look funny to one used to other interpolation methods, but are nonetheless undesirable. A way to overcome this is to smooth (or blur) the coefficients before each scale interpolation, so in the end the square blocks have diffused to some extent, the more so for lower frequencies. In this project, a standard convolution filter was used. Its coefficients and effects are shown in Schema : We can see the artifacts have disappeared from the second height map.

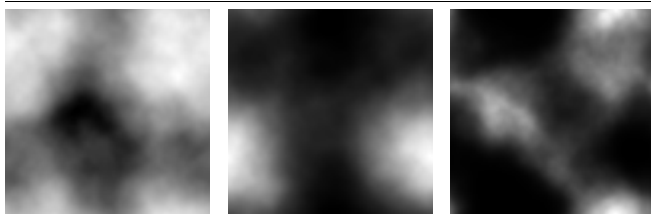
Now comes the problem of choosing a frequency distribution. Indeed, if noise at all frequencies were added with the same weight, the result would look very much like white noise and would be much too irregular for a realistic terrain. Thus, scaling each spacial frequency by a significant ratio is necessary. But how to choose this ratio? For this project, I have used the notion of Persistence, introduced initially by Benoit Mandelbrot. In short, it corresponds to an exponential decay of the scale ratio value with respect to the frequency. It can be seen as a measure of how smooth the final result will be (See Schema 4). As often, the quality of a model can be measured with how little (not how many!) parameters are necessary to tune it. In this sense, the Persistence model is a very good one as it allows to generate a distribution for any number of frequency levels (thus any terrain resolution) with the same only one parameter! It is also very well adapted to fractal analysis, and complements the scaling coefficient used to preserve energy in wavelet reconstruction, so was indeed a well suited choice for this project.



Schema 4: Effects of the persistence parameter

Finally, the last trick was to increase the height differences, so as to produce large plains and sharper mountains instead of uniform hills. This can be simply done by scaling all the parameters between 0 and 1, and taking the exponential of the values with a chosen exponent. This is the second parameter of my terrain model, and its effect are presented in Schema 5.

Now that we have a suitable height field, it can be scaled along its 3 dimensions to accommodate a given world size. Let's apply a color model to it. To make it simple, I used a grass color for the bottom, a rock color for the top, and an earth color for the slopes. Each vertex receives a color according to its height,

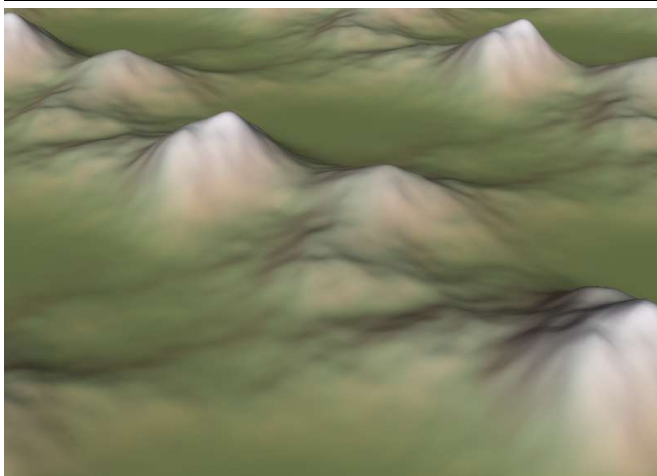


Exponent=1.0 Exponent=2.0 Exponent=3.5

Schema 5: Effects of the exponent parameter

and the cosine of the terrain normal at this point with the horizontal. A first linear interpolation is done between the bottom and top colors, then a second between this value and the slope color using the aforementioned dot product.

Finally, the vertices were meshed in triangles by splitting each height field quad along the higher diagonal. One possible result is shown in schema 6.



In this screen shot, we can see the terrain is perfectly tileable. In order to achieve this cyclicity, care must be taken in all steps of the construction: wavelet interpolation, smoothing filter, and normal averaging.

Schema 6: An example of terrain, with 2 cycles.

Simulation & physics

Now that we have a world, we also need rules to define how the the agents can act on it. OpenSteer uses a point-mass model, and limited Newtonian physics. In this project, we'll extend it somewhat, but with performance issues in mind: Physics have to be applied whatever the AI, at a high frequency (see the Simulator section later on), and should therefore be as fast as possible. The goal here is not to provide an industrial level physical simulation: we need a realistic looking application, not a faithful representation of reality. Some effects like the propagation of waves, or material properties, are included in some games but not considered here: While providing added value for realism, these have little if no impact on AI and therefore their cost/benefit ratio was deemed too high for this project.

On the other hand, I extended the simple point-mass model with the following features:

Energy consumption. It determines the ability of an agent to apply some steering forces. This is especially important in a prey/predator scenario for example: when a bird has no more energy it falls down with gravity. For a race game when a car has no more fuel it stops. On the other hand, this project makes the assumption energy is always necessary to apply a steering force. This may not be true for example in the case of a car which can still turn with its remaining speed even when no fuel remains. This effect clearly has an impact on AI, but is also usually only transient. Thus it's estimated not to be a big loss for this project. Moreover, the rotational momentum hack (see below) has the side effect of making a minimal move still possible along the forward direction. A future extension of this project could be to remove this side effect and introduce the notion of minimal autonomy.

A drag coefficient to slow down moving objects. This part still need further improvement as for now it doesn't take in account the difference between the air and the floor. This effect was introduced mainly to slow down moving objects with no energy, and make this state even more transient. As said, implementing a frictional slowdown model for the floor would certainly be an interesting extension to this project.

Energy storage. This was introduced to model the fact some mass is necessary to store energy in some cases. For example, a car using fuel has limited storage capacity, and the fuel itself increases the car mass. A predator can only eat so much, and also has a mass increase. On the other hand, a solar car has infinite energy capacity and need not store fuel. In this project, the energy limit and mass storage conversion ratio are taken into account.

Maximum speed and steering force length. These were already present in the OpenSteer library, and represent the physical limitations of an agent.

Gravity. As the drag coefficient above, this is not really part of the agent model but rather of the environment. Its effects are most noticeable for flying agents. An interesting side effect of gravity is to allow an agent to reach a speed higher than its own maximum limitation, which can be particularly interesting for predator birds AI.

A rotational momentum hack. A trick is used to model rotational momentum, not physically based. It has little impact on AI (see the remark in "energy consumption" above), but it has a

very low run-time cost, and increases realism by making heavier agents turn more slowly on themselves.

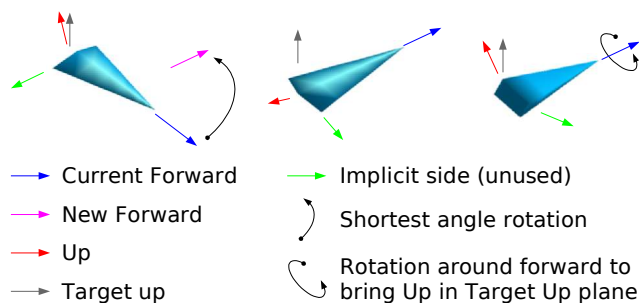
Attitude correction

By taking into account all the effects presented above, it is now possible to compute the effects of the steering forces applied to the agent. As a result, the agent will have a new position, and a new forward direction. For this project, I re-use the notion of an Attitude Quaternion, widely used in the spacial industry⁴. This allows to fully specify the local to world-space transform, with only a few parameters: position, rotation center, attitude, and scale. For this project, the rotation center (in local space) and scale are taken into account, but are fixed in practice. We now have only two parameters to worry about: position and attitude.

Position is easy to update. It's just a matter of converting forces to an acceleration, integrating it into a speed, then speed into a position difference, and adding the old position.

Attitude, on the other hand, represents the direction of the local space forward vector in world space, together with the amount of rotation the local space has around this forward vector. The new forward vector is given by the previous integration. The problem comes from the fact infinitely many rotations can map the old forward vector to the new one, and we must choose only one.

One possibility is to choose the angle rotation to map the old forward vector to the new forward vector. This can be done with a quaternion transform using only fast geometrical properties, and no expensive call to trigonometric functions or their imprecise table lookup approximation. Since we have to apply this transform at each integration update, this is a very desirable property!



Schema 7: Attitude correction

An additional possibility is illustrated in Schema

⁴ See <http://logiciels.cnes.fr/MARMOTTES/marmottes-mathematique.pdf> for example, for a presentation of many mathematical concepts used for space navigation.

7. It consists in maintaining an Up vector in addition to the forward vector, and add a second rotation around the new forward vector. This second rotation does not modify the forward direction, and brings the new Up vector in a plane defined by the new forward and a target Up vector. By setting this target Up vector to the floor normal, the effect achieved is for agents to always match the local floor horizontal. Hence they smoothly follow the floor curvature as they move.

Care is also taken for reconditioning the quaternion after too many rotations, when this becomes necessary. This is a common problem appearing with matrices, where after many matrix multiplications the numerical errors accumulate: The final rotation does not match the intended one and the object scales down.

Quaternions are slightly less sensitive than matrices as they use less multiplications for rotations, but are certainly still subject to numerical error roundups. And though less frequent, the errors are more visible: the objects tend to shear and flatten. Thus, in this project, I detect when a quaternion is too badly conditioned, and it is renormalized. If additionally the transform of the local space forward doesn't match the world space forward with a good enough precision, the quaternion is fully re-built and not just renormalized.

Thus, the agents always have the right orientation no matter how long the simulation runs, and it is still possible to use the fast incremental quaternion update to compute the new attitude.

Simulator

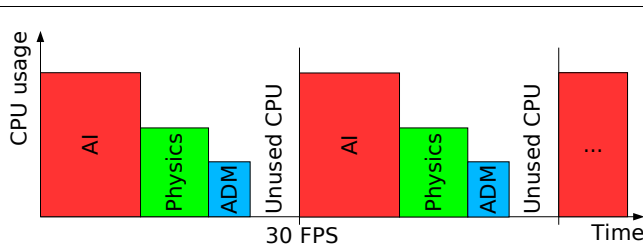
A simulator may not be the easiest thing to implement, but it is a very useful tool once you have it. In the context of this project, by simulator I really mean the basic version: a scheduler to organize events, and logical time facility.

Some major advantages it provides are:

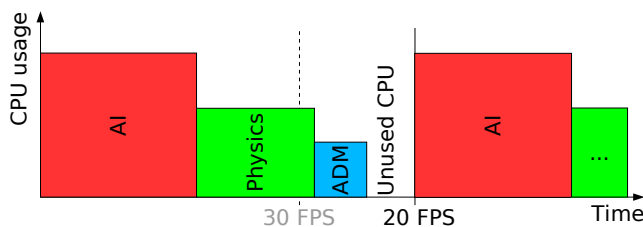
- ◆ The ability to separate the frequencies of AI, physics integration, and administrative tasks. Indeed, AI does needs not be applied as often as physics integration, and garbage collecting for example may be invoked even less frequently.
- ◆ The ability to spread the events in time so as to best use the CPU resource, and to allow graceful degradation at constant frame rate when the system is overloaded (see Schema 8).
- ◆ The ability to run the program at a different time scales. It's possible to pause and restart the simulation, to slow it down and analyze

what's going on, or even to run it faster than real-time to study long-term effects if the CPU consumption allows it. By extension, when the simulator is paused, it's possible to serialize the objects to permanent storage and "save" the whole simulation (not implemented in this project, but worth a note).

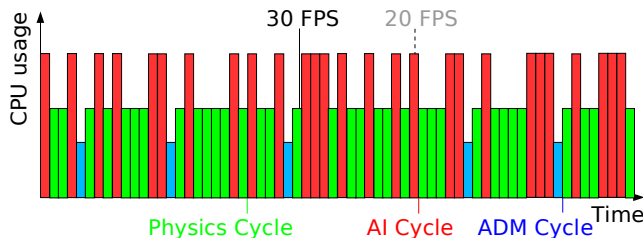
- ♦ The benefit of a fully deterministic ordering of events, which is especially useful in multi-thread programming. Together with an explicit random seed, this contributes to making scientific experiments reproducible.



Classic CPU waste effect while waiting for a full frame



Frame drop effect when waiting for the next vertical refresh rate divisor. In this example, a vertical rate of 60Hz leads to 30 and 20 frames per second. If the computations can't fit in one frame, then the frame length is increased to the next divisor. Not only can this result in a drastic drop in frame rate, but CPU is still wasted.



Using a simulator allows to spread the events in time, with separate cycle lengths. If all agents can complete their physics during one frame, the visual result will still be smooth. Otherwise, only those agents which could not complete their physics won't have moved during this frame. This is a graceful degradation case, and may even be unnoticeable if the number of agents skipping a frame is low enough. Moreover, these agents have priority on the others to do their physics on the next frame, so the degradation is statistically diluted over all agents in time. AI and administrative tasks don't have to be applied as often as physics, and have little if no visual effect, so can tolerate an even worse fate. Finally, no CPU is wasted waiting on the next frame, provided the graphical and simulator threads have well-defined protected sections.

Schema 8: Attitude correction

Additionally, it is possible to fall back to the first case of Schema 8 if desired. Why would this be useful, when spreading events is presented as an

advantage? In the case the number of agents is really high, and when the physics integration step is very simplified, the cost of posting and handling the events in the simulator may become of the order of the cost to process the events themselves. In this case, it may be better to regroup the physics of many agents in only a few events. The extreme case of only one event for the whole physics is equivalent to the first diagram in Schema 8. The same is also true for AI, but such cases are only meaningful in very special situations. Usually, it's much better to add some jitter when posting each event so they are spread over the whole cycle, and benefit from the progressive degradation effect should it happen.

Collision avoidance and detection

As pointed out in [1], collision avoidance and detections are 2 separate tasks. Avoidance is the problem of finding a steering force to prevent the collision, and detection is the problem of determining whether two agents are occupying the same space or not.

The collision avoidance routine of this project is initially based on the OpenSteer library ideas, but uses the OpenSceneGraph intersection routines instead of bounding spheres. After some experimentation, it is still unclear whether the benefits of real geometry intersections is worth their extra cost over ray/sphere intersection. More advanced tests could be an extension to this project, especially on the apparent behavior of the agents in each case, and to make statistics on number of collisions avoided or not.

Whatever the intersection method chosen, collision avoidance makes use of neighborhood queries to begin with. Now that I've presented the simulator, it's worth mentioning that collision avoidance needs only be applied at AI frequency, to produce a steering force. Since locality queries is usually the main CPU consumer by far, this is a very big improvement compared to frame-rate based simulations!

Collision detection is on the other hand necessary at each integration step. Fortunately, no new neighborhood query is necessary: Assuming the AI frequency is enough to capture changes in the neighborhood allows to use only the current list of neighbors for distance comparisons. Of course, if an agent comes from far away in less than an AI cycle it won't be detected. In mixed mode, several AI cycles may even be necessary in the worse case where none of the agents are in the locality database, and they need a few updates to find each other. However, thanks to the maximum speed limitation and by always including at least the obstacles in the locality database, collision

detection is reliable enough and very fast in this project.

What to do when two agents collide is another matter, and depends on the scenario.

Scenario

Four applications were developed in this project. They are presented below in order of complexity.

Swarm

This basic application mainly tests the locality algorithm, but also the physical integration routine, the shortest angle attitude correction, collision detection, and a very basic AI.

All agents are included in a non-cyclic world, without terrain. To avoid agents from leaving the scenery altogether, isolated agents wander around randomly with a configurable bias toward the origin. The final visual appearance of the swarm strongly depends on this bias value: the larger it is, and the more compact is the swarm.



Schema 9: An example of swarm with 1000 agents.

When in the presence of others, agents adopt a boid-like behavior: The AI combines steering forces to avoid neighbors, move in the local group average forward direction, and move toward the local group center. Only visible agents are taken into account. This has the side effect of making the leader of a group follow the wander/bias behavior. Depending on whether another agent takes the lead or not, groups usually don't wander too far away before coming back home.

Collision detection is done in the integration step, but can optionally be turned off. When a collision is detected, the agent bounce back. For added realism, it also keep some component of its previous forward direction, and of the

colliding agent forward too.

The result is a cloud of agents moving in group patterns, with sometimes trails of agents between groups. Its full 3D structure can only be apprehended by running the program, Schema 9 shows the entire cloud from some distance. It is reminiscent of natural gnat formations.

Colliterra

This stands for collision and terrain. This program extends the swarm by making the world cyclic, with a terrain, and with obstacles. The cyclicity allows to remove the bias toward origin, and thus we no longer have a cloud or global group formation.



An example of the colliterra program. In the foreground bottom-left corner we can see an example of critically bad collision avoidance when landing. This may be due to the fact the cyan agent was moving too fast due to the effect of gravity, and detected the obstacle too late to avoid it with the steering force limitation. Collision detection was on the other hand correctly handled (the agent bounced back at simulation re-start). We see also see that the yellow-green and red-magenta agents (in the same image corner) follow the local floor curvature. In the background, we can see the same group and trail patterns as in the swarm example.

Schema 10: An example of the Colliterra program

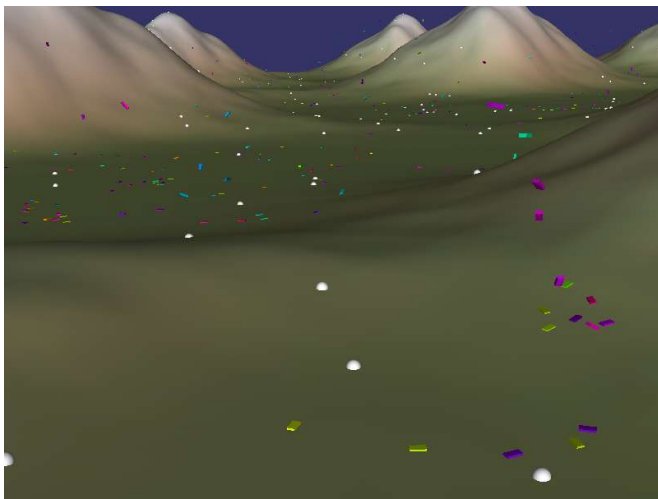
The AI reuses the idea of random wander when isolated, and boid-like behavior otherwise. Obstacle avoidance is also taken into account and takes precedence over the boid-like behavior.

In addition, all agents have a small probability to completely ignore this AI and take off. They then continue to soar up to a certain "safe" altitude where they resume boid-like behavior in full 3D. This allows to test the effect of gravity, which can be changed at run-time. Including the effect of faster-than-own-maximum speed, which appears when the agent falls down back to the ground.

All agents have an infinite energy supply, and

will pursue this AI indefinitely. Long runs have shown agents behavior are stable and this regime is reached in the few seconds after the simulation start.

As previously mentioned, the agents maintain their Up vector aligned on the floor normal. The result is visible in Schema 10, where agents always maintain their local-space horizontal correctly. The effect on flying agent is visible in Schema 11: the up vector is kept in the floor normal and forward vector plane.



Another example of the colliterra program where an agent decided to take off and other agents follow its lead.

Schema 11: Agents take off in the colliterra program

Predator

Predator extends the AI in colliterra to introduce the notion of energy. Agents now consume energy when they apply steering forces, and the only way to renew it is by “eating” another agent. Hence the name predator match: the agents in any one specie can predate on all the other species. Of course, when no energy remains, the agent has no choice but to become a passive prey...

The AI is now divided in two parts: The first part is the same as in the colliterra program when the neighbors are only friends (or in the isolated agent case).

On the other hand, when an agent of another species is detected, the reaction is as follow: If it approached the prey from behind, it chases it in the hope of not being seen. If on the contrary the agents are facing each other, they try to avoid engaging in a death match.

This is reflected in the collision handling routine: When two agents collide, the winner is determined as follow.

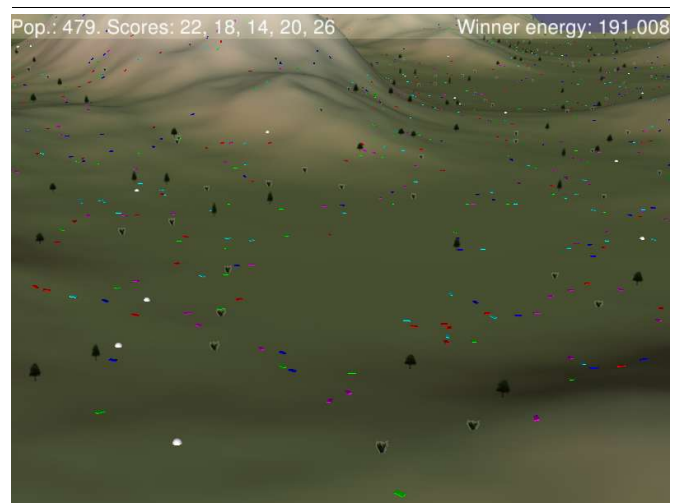
- ◆ If one agent catches the other from behind, it

unconditionally wins. Remember agents only “see” what's in front of them, so this case represents the situation when the prey hasn't seen the predator coming.

- ◆ If agents are facing each other (their forward vectors dot product is negative) then a death match is engaged. The agent with more energy wins the match, but loses an amount of energy equals to what the other agent had. This models a situation where the prey reacts with all its remaining forces, and the predator has to fight. If a draw occurs, both agents die.

The winner of the match then “eats” its prey by converting the prey mass into energy according to the energy footprint ratio, and its maximum energy capacity.

The species scores are monitored together with the winner remaining energy.



In the predator match we still see the groups and trails patterns, but usually a specie dominate in each group. This seems logical given the little chance an agent has to survive when surrounded by an hostile group. On the other hand, the limited AI applied in this program does not lead to the emergence of a prey group behavior, as we see in nature, when the preys gather to be stronger and share information.

Schema 12: Agents chase each other in a predator match

How this program evolves in the long term depends on the initial conditions. When the energy footprint ratio is too high or the energy limit too low, the agents can't store enough energy and it runs out before they are able to catch a prey. Then, we have a collection of passive bots slowly moving on in their forward direction. This problem appears when the population goes down to below a threshold, where chance encounters are too limited. This threshold also reflects the fact no advanced predatory behavior was introduced in the AI, but it largely depends on the energy parameters in any case.

On the other hand, when the agents can collect and keep enough energy, the “chance encounter

is not enough" population threshold is very low. In this case, the agents have more time to wander around before they run out of energy. We also see more agents can afford to fly, which they could seldom do in the previous case. Note that the AI has not changed, it's just that agents taking on flying in the first case were very soon running out of energy and brought back to the floor the hard way...

This project framework offers lots of extensions and possibility for testing various AI. Extending and creating new AI routines may thus be considered in the future. A particularly interesting extension was tried with neural networks, in the hope of getting a good starting point for a further genetic algorithm. This is detailed in the next section

Crogai

This acronym stands for Crowds, GA, AI, and reflects this initial intention.

The main problem with genetic algorithms in this project context, would be to find a good starting point. Indeed, agents taken with random initial conditions have little if no chance at all to predate and survive long enough to reproduce.

Thus, the first step is to provide them with a good AI to begin with, and the second step would be to evolve this AI.

A candidate choice for the machine-learned AI is neural networks. And within neural network, a quite limited but simple model is the two-layer perceptron. This model is also quite fast, which is a desirable property if it is to be applied each AI step for a population of agents.

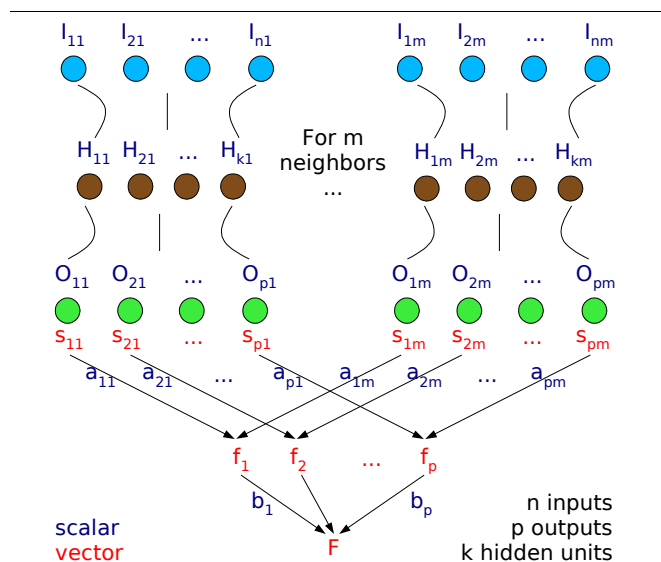
The main problem now is to design this neural network in such a way as to be applicable to this AI. In other words, this amounts to finding good inputs, an output to steering force mapping, and a good error function. A way to account for variable number of neighbors must also be found. The architecture retained is presented in Schema 13. Once the network is built and ready to be trained to minimize the error function, it needs a teacher. Any explicit AI written within this project framework can be used as a teacher, for example the one in the predator game. The network will then try to learn what this teacher AI does.

Inputs

For each neighbor agent, the network is run with the following inputs. Parameters marked 'c' are common to all neighbors computations, but provided nonetheless on a per-neighbor basis.

Parameters marked 'n' are neighbor-dependent, and set to respectively 0,0,0,-2 when there is no neighbor. Parameters marked 'e' would only be used during the evolution phase, and not the training phase. They represent work in progress, not yet included in the project. They are meant to represent the acquired knowledge part of the agent, and are reset when creating a new offspring.

- ◆ c: number of obstacles
- ◆ c: number of friends
- ◆ c: number of enemies
- ◆ c: altitude
- ◆ c: remaining energy
- ◆ c: mass taking in account energy storage
- ◆ n: squared distance to target
- ◆ n: direction of target relatively to our forward direction. Is the target on the side?
- ◆ n: target forward direction relatively to this agent forward direction: Are we in front or behind the target?
- ◆ n: specie of the target. -2 is a flag to indicate no neighbor instead of a specie, -1 is reserved for unknown objects, 0 for obstacles.



The network is applied to each of the m neighbors with the corresponding n inputs. This produces m outputs, which are factors for the m corresponding per-agent steering forces. For each force, the agent results are aggregated according to a chosen policy: Averaged, or Larger norm takes all. This gives a unique f vector for each steering force. These f vectors are then once more aggregated with a second policy choice, into a final F force. This force is the final AI result.

Schema 13: Neural network architecture

The planned features for genetic algorithm were:

- ◆ en: message of the target to the world.
- ◆ ec: current message of this agent to the world.
- ◆ ec: memory (private message of this agent to itself) from previous AI update, index 0.
- ◆ ec: ...
- ◆ ec: memory from previous AI update, index number of memory - 1

Outputs

Each network output is a coefficient to apply to a vector. The vectors in question are given below, and markers 'n' and 'c' have the same meaning as for the inputs. Additionally, scalar values were planned to be produced during the evolution phase, but not included in the project yet. Those are marked 'e' and are meant to be used only after training (they are not part of the error function).

- ◆ n: seek target position
- ◆ n: flee target position
- ◆ n: avoid collision with target
- ◆ n: pursue target
- ◆ n: evade target
- ◆ n: position difference vector
- ◆ n: target forward direction
- ◆ c: wander direction (constrained random direction, see Agent.h)
- ◆ c: floor normal at this point
- ◆ c: our own previous forward direction

The planned features for genetic algorithm were:

- ◆ ec: new message of this agent to the rest of the world
- ◆ ec: new memory (private message of this agent to itself) for next AI update, index 0
- ◆ ec: ...
- ◆ ec: new memory for next AI update, index number of memory - 1

The neural network is run for each neighbor, then the computations are aggregated according to the following policies (see Schema 13). For each result vector:

- ◆ Average: average vectors from all neighbor computation contributions to get a steering vector. The a coefficients are all 1/m.
- ◆ Winner take all: Get the largest norm output * force vector as the only one for this steering force. The a coefficients are all null, except exactly p coefficients which are 1: the ones for the largest output result norms.

Once a steering force is computed for each output, the forces are then once again aggregated according to one of the following policies for the final steering force:

- ◆ Average: average contributions from all individual steering results. Each b coefficient is 1/p.
- ◆ Winner take all: retain only the largest norm individual steering force. All b coefficients are null, except the one for the largest norm steering result which is 1.

Averaging should in principle give smoother results and allows training all network weights.

Winner take all is supposed to give more natural

results (ex: avoid the closest), and may be how the teacher AI decides on only one clear action in a case where multiple choices could be made. Unfortunately as gradients are 0 for unused weights in this case, the training phase may not go so well.

Of course, both policies can be mixed. I tried all combinations for this project, but the results are not very good so far (see below).

Transfer function

The two-layer neural network also needs a transfer function, usually sigmoid-like. I devised one especially for this project:

$$f(x) = x / (1 + \text{abs}(x))$$

This function is sigmoid-like, -1 / +1 bounded, continuous to any order, and much faster to compute than tanh. It is also unfortunately slower to converge, see the note below.

Many neural network packages use tanh, to the point it has become the standard function to use. It has the very nice property of having $\text{tanh}' = 1 - \text{tanh}^2$. Thus, backpropagation can be made faster by reusing previous computations for the derivative, and very often the goal is to best train the network so this is a very desirable property. Others use the sigmoid function $s(x) = 1 / (1 + \exp(-x))$, where $s' = s(1-s)$, for the same reason.

However, in the long-term goal of this project, the neural network is intended to be used primarily in forward mode, and backpropagation / explicit training is only a way to provide a reasonably good starting point for the genetic algorithm to work on.

Thus, the most important point is to have the fastest transfer function: we are going to apply it a lot, number of hidden units plus number of outputs times per agent, and for many agents! On the other hand, it doesn't matter much if we couldn't re-use some computations in backpropagation: the learning phase is done only once, and is fast enough as it is with the function above anyway.

As a matter of fact, with the function above, we can reuse the results exactly the same way as for tanh: $f'(x) = 1 / (1 + \text{abs}(x))^2$, so:

$$f' = (1 - \text{abs}(f)) ^2.$$

Thus, even the training phase is fast.

Note: To moderate this finding, numerical experiments have shown that this function is much slower to converge while training than tanh, about the order of what the sigmoid does. Thus, if all you want is fast convergence, tanh is

a good choice. On the other hand, since we don't require exact convergence as all we want here is a good starting point for the genetic algorithm, we don't care much. Also, many people use the sigmoid, and this function is about the same convergence speed, so it's really a matter of taste I suppose.

Error function

During the training phase, the error function is computed as follow:

$$E = 1 - \frac{\vec{F} \cdot \vec{T}}{|\vec{F}| |\vec{T}|} + \frac{1}{2} \frac{|\vec{F} - \vec{T}|^2}{|\vec{T}|^2}$$

Where F is the result force obtained from the network by the method previously described, and where T is the target force to learn from the teacher AI (this is what the teacher does for itself in this situation).

I devised this error function especially for this project: it includes both a directional and a normative component. Additionally, it makes the error landscape smooth around each minimum. Thus, I had reasonable hopes for convergence using this function.

Since this training phase is stateless, without memory, the neural network can be registered on many teachers at the same time to get data from many different situations. This allows to collect data very efficiently: 200 teachers running at an AI frequency of 5 times per second would produce 1000 mappings per second, and those numbers are not excessive given the current hardware.

The training is a very simple on-line gradient descent. Batch mode is used to propagate the error gradient back to each the neighbor individual network computation. This means the error gradients are averaged on all neighbors, since the backpropagation in fact acts on the same network repeated m times. An extension to this project would be to try other ways to compute the error derivative with respect to each network parameter.

Analysis of a Failure

The results are quite disappointing. The network doesn't converge at all, no matter which transfer function is used, and no matter which simplification is done to the error function.

I suspect the main reasons are:

- ◆ The same network is also used when there in no neighbor. But in this case, many parameters are meaningless, and the teacher

AI does use a completely different action. This point should be corrected in a future version of the project.

- ◆ It may be that the mapping is too discontinuous. Multi-layer perceptrons are supposedly universal function approximators, but this is assuming an infinite hidden layer size. In practice, when the function to approximate is too discontinuous, the network fails to find a good approximation. I have already encountered this very problem a few years ago, in another context, so I know its effect are not to be neglected.

A final word

I had a lot a fun programming this project, and I also learned a lot. I hope it will continue to live as I distribute it on the Internet, and that people will enjoy it as much as I did.