BACVA3

Innovations Project

An AI Module System for Real-time Games

[xymosBot system]

by

Laurie Hufford

Introduction

When this project was started it was intended to be a system by which a user could specify routes around a map, along which a given number of computer controlled characters (bots) would traverse, the results of which would be saved to a file. This data could then be imported into Maya using the MEL scripting language and used to generate a number of motion paths that 3D models would be animated along. The program would have had a GUI that would allow the user to control every aspect of program. In the process of producing this a lot of design work went into the GUI. When it came to designing the AI that would move the bots, however, it was realized that this was a fairly huge subject in it's own right, one that the author has been fascinated by for a long time. It was at this point that the project changed direction, as EITHER a very simplistic, non-expandable AI model with a user interface and connection to Maya OR an open-ended experimentation into what was possible with AI in terms of real-time game applications could be produced. The expandable game system was chosen, as it would provide a lot more scope for improvement and innovation, and be a lot more enjoyable project to produce.

The following report will attempt to describe the concepts behind this implementation and some of the methods used to create it. The model for this program can be broken down into three sections: Structure, Locomotion and Thought (Figure 1).

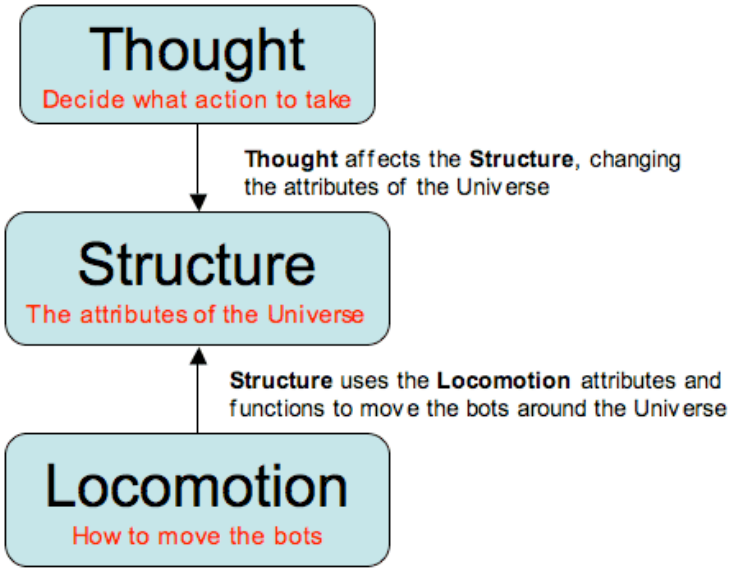

Figure 1 - The connections between the concepts of Thought, Structure and Locomotion

The main area that this project is interested in is the Thought section, as this is what drives the bots and is the most complex of the three. It cannot, however, exist on its own, and both Structure and Locomotion are vital to the functioning of Thought.

Structure

To implement any kind of artificial system, be it for recorded simulations or a real-time game engine, a structure has to be set up that will allow simple access to the inner workings of the world that is created.

Although it isn't the focus of this project, the structure that is behind the implementation of this code is absolutely vital to its successfully operation.

Using C++ meant that every object within the program could be specified as a class in its own right, which made visualising the structure a lot simpler.

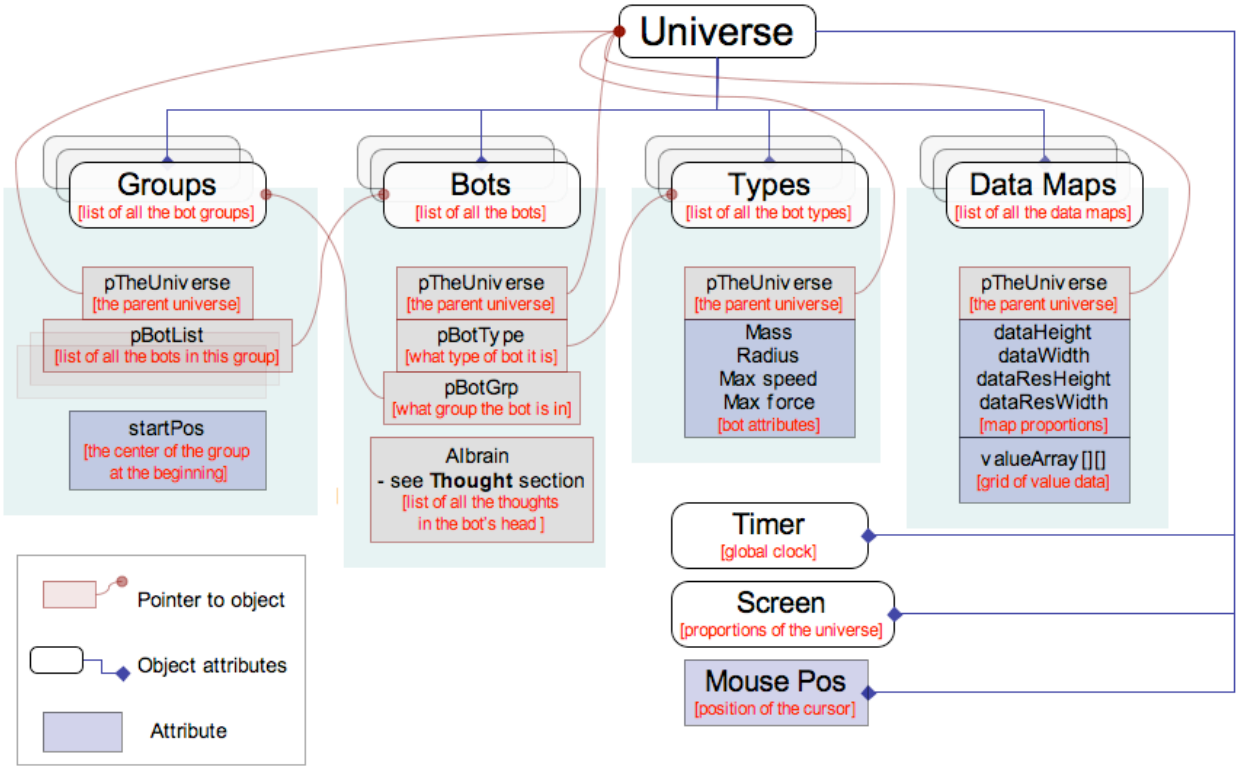

Figure 2 - Structure of the objects and their relation to one another

Objects

The Universe (universeclass.h)

The Universe is the world in which all action is taking place. It stores the data for all of the elements that are used in the system, the Bots, Groups, Types and Data Maps. It also stores the basic information about the world that we are in; the size (in Global Units) of this world and the clock that time is taken from.

This structure of a parent object that holds all the other objects is useful for sharing data between objects. As every object has a pointer back to its parent universe it can call any object that is within that parent.

Initially when building this system I did not have a Universe class implemented, with every element floating separately. I knew that at some point it would be necessary to have some cohesive element between each object, but was too focussed on the other aspects. This is a good example of where preplanning is paramount in programming, as when I began putting the Universe class into the system a large portion of previous code had to be rewritten to accommodate it.

For the initially proposed job of moving artificially controlled bots along routes this Universe class binding all other objects works well. For the evolved purpose of this project, an Artificial Intelligence system for game engines, I am not so sure that it will be as perfect. It would require the entire game to be built around this structure, rather than fitting into any other game engines that require it. As at this point I think it will only be me using it in games, I don't think this will be a hindrance, but for future uses it may have to be redesigned to some extent.

The Bot (botclass.h)

The name 'Bot' is used in this system to describe any object that can move, and in its movement display simulated thought processes. The name describes what are referred to in other AI work as Agents or Boids [Reynolds 1987], or in games as NPCs (Non-Playable Characters).

The Bot is a type of Vehicle (an even more generic object that deals with position and direction – see **Locomotion** section), and as such inherits the attributes and methods from this superclass.

Every Bot has a group (pBotGrp) and a type (pBotType), each storing information necessary for the full functionality of the Bot. Group and type information is not specific to an individual Bot, they exist outside of the scope of an individual, and the Bot simply points at the type and group that it wishes to be part of.

Each bot also has a brain (more specifically an AIcluster – see **Thought** section), which stores a list of all the thoughts that the bot has in its head.

The Bot Group (botgrpclass.h)

A group is a collection of Bots. They do not have to be of the same type, they just deal with bots in the same group differently to bots in other groups. Every group stores a list of pointers to all the bots that are of that group. This allows for easier access to members of individual groups.

Groupings are also used to initially place bots in the same area. At the moment this is a very basic method that puts them in a straight line. In future versions a more advanced system for arranging them in formations or randomly around the specified point will be implemented.

From the original specification of the problem I wanted to have groups of bots all following the same route. A method of giving orders to every member of a group is not in the current implementation, but in the future will be available.

Another possibility for future improvement is changing the grouping system from being based on specific groups to being a variable that decides how strongly a bot aligns itself with a group, so influences from other groups can sway a bot's allegiances.

The Bot Type (botclass.h)

To allow for variations in the sort of bots that travel around this world a way of defining generic attributes was required. This is where the Type class is used. It allows for common characteristics to be given to multiple units. For example, if the user wishes for certain bots to act like tanks, with a large radius and even larger mass, a type needs to be created with the attributes correct for a tank. This can then be applied to any number of bots and they will act under the influence of the new type's traits. The same can be done with human qualities, and the bot will act like a human.

In the future I would like to implement a more individual-orientated attitude towards the bot's attributes. It would be possible to keep specific user-defined types, but when a type is assigned to a bot it randomizes the characteristics within defined limits. The attributes would be stored with the bot itself

rather than in a separate class, and as such could be altered individually depending on the bot's actions within the world.

The Data Map (map.h)

Data Maps are used to store information about the landscape the bots are traversing. This information can have many different forms, from terrain height to surface type and even fear and desire regions. The data can be taken from image files or updated on the fly (not used in the current implementation, but possible).

When importing data from an image the RGB information is read from a specified file and averaged so that only the luminance values are used. The images can be any size, and the data is enlarged to cover the dimensions of the universe. Image data is then scaled so that it is consistent with the type of data it represents. For example, if an image representing the height of the terrain is imported its values will initially be between 0 and 1, so it is scaled by a maximum height value to keep it in proportion with the rest of the program

As can be seen in Figure 3 any fidelity of data can be used. A map stores both the actual number of data elements and the size of each data element in Global Units (GUs – see **Screen** section below). The map also provides methods for converting between GU coordinates and data element coordinates, and retrieving the values of data elements.

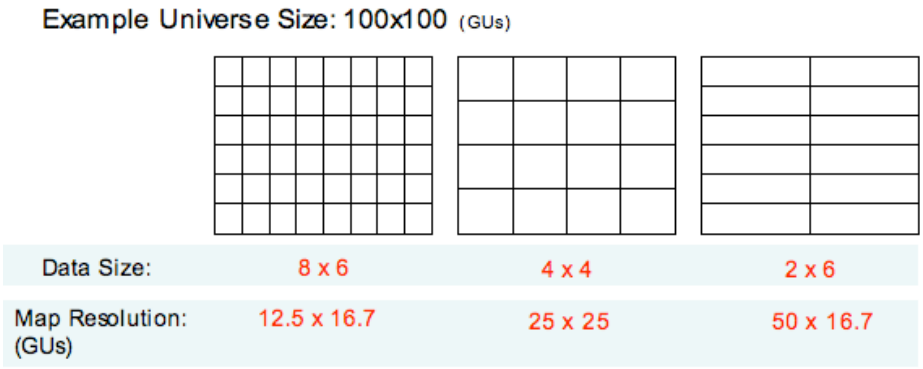

Figure 3 - Variable Data Map fidelities

This ability to have variable fidelity maps is very useful as it speeds up data retrieval. Something that would be of great benefit in future versions would be the inclusion of variable fidelities for individual maps, so a high detail and low detail version of the same map is stored, and different versions are accessed when appropriate (for example, when a bot is off screen it need only use a low detail map, as it's movements are not being scrutinised).

The bots use map data for their **Route Finding** thought unit (see **Thought** section), but it can be utilised for any number of other functions. For example, the strength of a group could be stored in a map and updated at run-time. This way bots can decide whether to attack another bot based on the strength of that bots group in the area.

Another improvement that could be implemented in the future is that of giving each bot its own map that specifies what regions it has seen. Any actions the bot takes using map data will be limited to the regions it has seen. It could then be possible to share knowledge of the landscape between bots when they meet, giving each bot the view of the world that the other bot has had. When combined with AI this could lead to fairly realistic actions on the part of the bots.

Not all data in maps has to be image based or generated on the go. Other options for map data could be information from 3d objects such as OBJ files (vertex positions converted into a height map, for

example) or storing map information as simple polygonal objects, such as circles, poly lines, or filled polygons. To not have to rasterize the world information would make storing data on small objects such as chairs or thin fences a lot more convenient.

Timer (timer.h)

The timer is a clock class that was created for this project. It can be used in any program, and allows useful controls over time. It can be paused, stopped, started again, and the time can be updated manually, so the interval between updates and the number of updates per second (or Frames Per Second) can be returned.

An interesting possible use for this timer is giving every bot its own internal clock, so that it can measure durations and generally time events. Timers could also be generated at runtime to keep track of things such as camera moves or to stop loops from executing forever.

Screen (screen.h)

When the Universe is created a size is defined, in Global Units (GUs), which is the maximum width and height of the world. A Global Unit is a single unit in the Universe's coordinate system.

The Universe has a globalMap attribute (the screen class) that stores this information. It also gives access to conversion methods that can convert from GUs to the openGL specification of the screen (-1,-1 in the bottom left; 1,1 in the top right), and also from the monitor's pixel position to GU coordinates (useful for using of the cursor's position in the program).

Giving this GU scale to everything provides a degree of separation between the objects in the world and the programming behind the world. It also allows the world to be thought of in any measurement system (kilometres, metres, miles, millimetres etc). We can use the same programming to define a universe the size of a tabletop with bots the size of ants or a universe the size of a solar system, with bots the size of planets.

Mouse Pos

To allow for objects within the Universe to access the position of the cursor a Vector to the mouse's coordinates is updated every time the mouse moves. The coordinates are converted into GUs so that objects do not have to think about the world outside of their Universe.

As part of a game engine this would be handled by a completely different section of the program, but for the sake of debugging it is very useful. For the original project design the user would specify a rough route for the bots to take, in which case the mouse would be required.

Locomotion

Now that we have structure to the world within the program we now need a way of moving the bots around this world. The purpose of separating the locomotion methods from the structure of the universe is to allow different types of locomotion to be used without having to rewrite any of the rest of the program. In the current version a simple *"idealized vehicle"* [REYNOLDS 99] has been used, but in other version a more physically modelled vehicle could be applied.

The design of this simple vehicle is taken from [REYNOLDS 99]. It provides a generic base from which a lot of different types of transportation can be simulated. Although not realistic this is just a simulation (and accurate movement of the bots is not the purpose of the project), and as such will be adequate for our needs.

Every bot is defined as having a position, a velocity (a vector from the position to where the bot will be in one seconds time) and a mass. Movement is generated by applying a force to the velocity (taking into account the mass, so that acceleration is simulated). A new position is calculated every clock tick by adding the velocity to the position.

The forces that are applied to the velocity are what control the movement of the vehicle. This is what the AI engine generates when it thinks of movements that it wants the bot to perform.

Both the velocity and the applied force are given maximum values so that they cannot increase to infinitely. To give a more realistic movement to the bot an asymmetrical maximum force could be used, as in Figure 4 (not implemented in the current version).

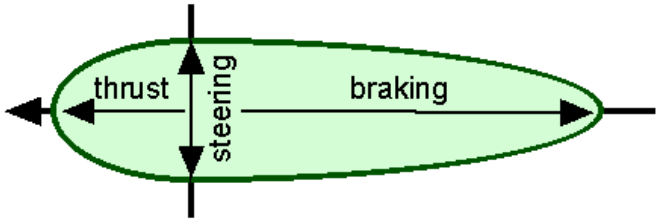

Figure 4 - Asymmetrical steering forces [REYNOLDS 99]

In this program a 'vehicle' class (vehicle.h) has been defined and any object that we want to move must be a child class of this vehicle class so that it inherits all of the attributes required to perform the movements. As this implementation uses botTypes to specify the traits of the bot the mass, max speed and max force are stored in the type. This does not keep with the original idea of a separating the structure from the locomotion, and is something that needs to be altered in future versions. Because of this design error certain movement functions have to be defined within the bot class rather than the vehicle class.

Thought

Overview

Controlling the thought processes of a computer-generated entity is a problem *"almost as old as computer science itself"* [ANDERSON 03]. Depending on the specific task in hand there are many approaches to solving the problem. Although the method used in this program is very much the basics of game AI, the implementation will hopefully allow for the integration of more disparate Artificial Intelligence systems, such as evolutionary techniques (in which the brains of the bots are evolved over time) and decision trees (where the bots store connections between every action and outcome, and as such 'learn' to make the best choices).

In the world of game AI a rough approximation of the actions a human would take is all that is required to fool the player that the their opponent (or team-mate) is acting with more innate consciousness than the actual coding belies. The point of the AI engine in a game is to provide levels of abstraction that a computer program (and a level designer) can understand.

The method of abstraction that has been used in this implementation is one that is believed to be new (or at least a hybrid of other approaches – no method is truly new, but that is a whole other discussion). The author not read descriptions of every implementation of AI in every game created, so it will not be stated that this is an original system, just one that is not a direct copy.

In the applied method every action is considered an 'AIthought'. These need to be programmed by the developer to produce the desired results when performed. Making every action a single type of object means that a bot's brain activity is homogenized and can be represented in a single list, and performed without regard for the specifics of the action.

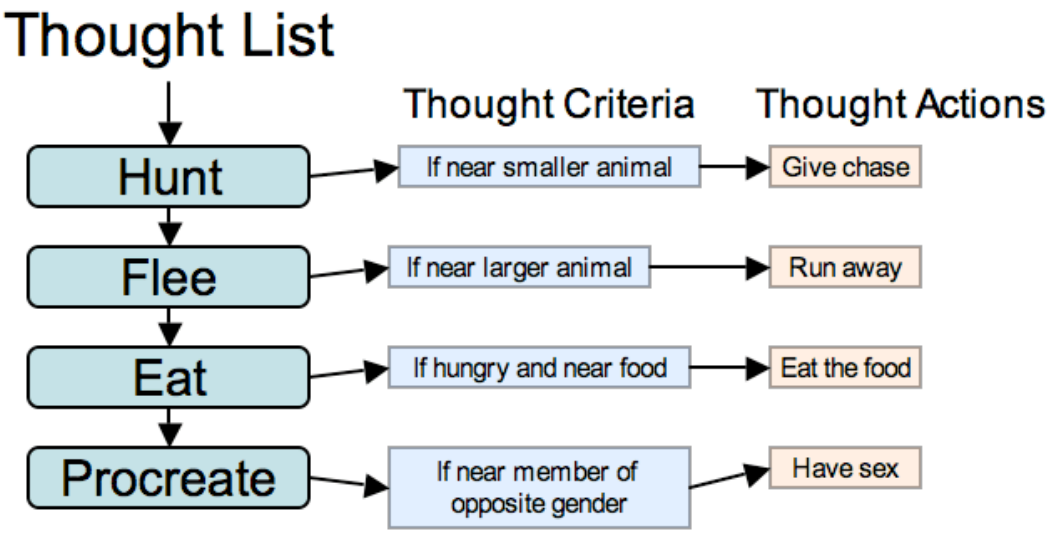

Figure 5 - Example of basic homogenized AIthought list

In Figure 5 the thoughts in the list are simple examples and the actions and criteria are vague generalisations. As we can see, however, it does not matter what the thought is or does, it's dealt with the same as all other thoughts.

Whilst these are simple examples they can still be broken down further into component thoughts. To hunt is the same as moving towards a point whilst finding the shortest route and predicting the prey's position. We need to therefore make even smaller thought building blocks that control motion and perform rudimentary predictive calculations. Any number of these building blocks can be programmed for a specific task so that they can be combined into a higher-level thought. Hopefully a library of these building blocks can be developed making the production of complex thoughts a lot easier.

We have the basic building blocks that every thought is made of, what we need now is a way of combining them in intelligent ways. This is where the 'AIcluster' is used. The AIcluster contains a dynamic list of thoughts that can be added to and removed from at the will of the AIcluster's programming. It is within the AIcluster that all of the criteria are checked and evaluated to give more specification to the actions that are added to the thought list.

The key to the open-endedness of this system is that the AIclusters are themselves classed as AIthoughts, so clusters within clusters can be created, giving endless combinations of thought building blocks and higher-level thought clusters (Figure 6).

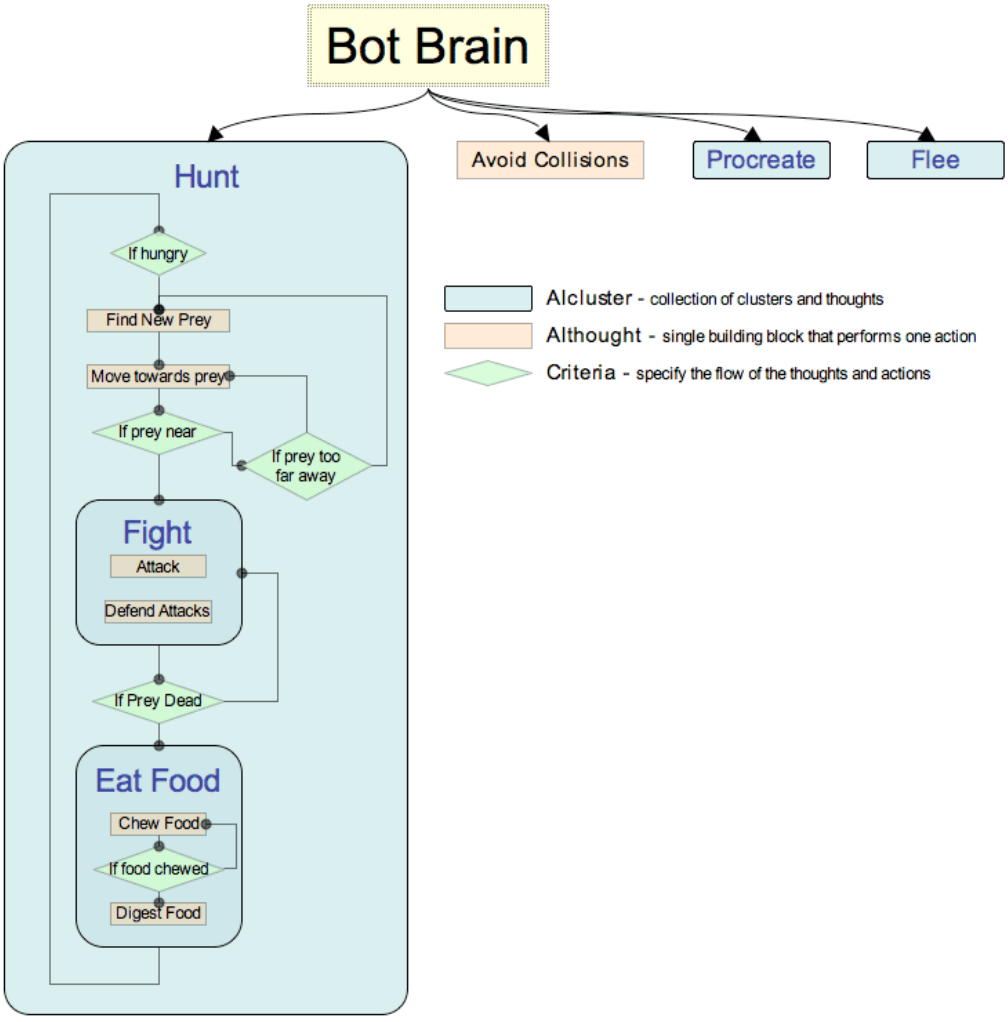

Figure 6 - Example of combining thoughts and clusters into higher-level thoughts

"They are components of a larger structure, like notes of a melody or words of a story. In order to make interesting and life-like behaviours we need to select among, and blend between, these individual components. Unless an autonomous character exists in an very simple world, it would seldom make sense for the character to continually execute a single steering behaviour." [REYNOLDS 99]

An issue that now arises from this system is the combining of different thoughts. If the bot has two move thoughts, each pointing in opposite directions, we do not want the bot to stand still or aim down the middle. This is an extreme example and should really be controlled by a parent cluster. Another, more realistic, example would be that of a bot moving towards a point and avoiding other bots whilst it does this. Both avoidance and moving are separate thoughts but must work together. To handle this a priority and weighting structure has been implemented.

To blend between thoughts a weighting value is assigned to every AIthought (Figure 7). The effect of the thought is scaled by this value, with a weight of 1 applying the thought fully, and a weight of 0 meaning the thought has no effect at all. How the thought is scaled is determined by the programmer, but for the example of movement in Figure 7 it is clear that the force with which the bot is moved is the attribute that is effected. For a cluster of thoughts every thought that is within the cluster inherits its parents weight. So, if a child thought has an initial weight of 0.5 and its parent cluster changes it's weight to 0.5, the child's new weight will be 0.25.

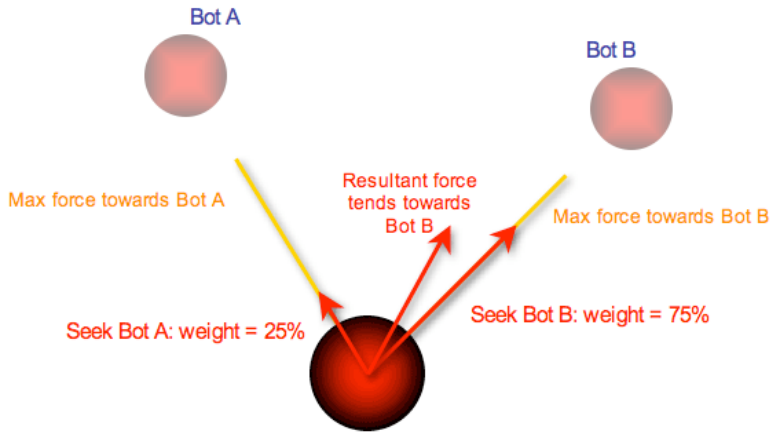

Figure 7 - Weighting of outputs from two thoughts

The ability to weight a thought does not give us total control of the thoughts that are called. We need to be able to only call certain thoughts based on their urgency. To deal with this problem a priority system has been implemented (Figure 8). Every thought has an integer value that represents its priority. Any thought that has a priority of 0 will always be called. Only thoughts with the next highest priority (in Figure 8 this is a priority of 1) are also called. Any other thoughts (Thoughts 4 & 5) are disregarded. This allows for urgent thoughts to override all other thoughts, except for instinctual ones, such as collision avoidance, which would be given a priority of 0. For example, if a bot is looking for food and a predator is detected nearby all other thoughts of foraging are ignored and escape is placed at the top of the priority list. The bot will not stop avoiding obstacles while it is running away, however, as these remain instinctual and priority-less.

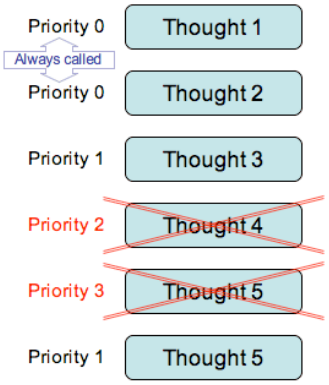

Figure 8 - Prioritized thought list (p0 always called, then next highest priority)

Whilst possibly this exact implementation has not been used before, it has been influenced by a lot of methods that are widely known. These include Finite State Machines (FSM), Hierarchical Finite State Machines (HFSM) and Fuzzy State Machines (FuSM). For descriptions of these AI solutions see [ANDERSON 03] or [FUNGE 99]

The method for a priority list and weighting values is influenced by [REYNOLDS 99] in which, under the 'Combining Behaviours' section, he describes a fairly similar approach.

Implementation

All of the AI thought coding is in the file 'ai.h', except for the routeFinding module, which is in 'astar.h'.

AI actions

To aid the homogenization of the AIthought system every thought is forced to contain four methods. These are 'action()', 'success()', 'failure()' and 'updateActivity()'. These methods are called by the parent cluster based on the 'activityState' of the thought. The activityState describes the mode the thought is in, analogous to the FSM technique. When activityState is 0 the thought is currently active, and the action() method is called every clock cycle. If activityState is set to 1 the thought is considered a success and the success function is called every cycle, and if set to 2 the thought has failed and the failure method is called every time. Any other value is ignored, so 3 or above can be considered an inactive state. This way, if a single action is required when a thought reaches a successful conclusion, it is possible to call that action in the success() function, and then set the state to inactive, so that the success() function is not called again.

The updateActivity() method is the criteria by which the thought decides it's activity. This method is not called in the parent cluster, so it is not necessary to use it. However, having the framework of a criteria method already set up allows every thought to be built in a similar fashion and keeps the code from becoming confusing.

Althought modules

All of the modules that have been implemented, apart from the RouteFinding algorithm, were taken from [REYNOLDS 99], a very succinct description of the many possible ways to steer an autonomous agent. Only a few of the methods have been implemented, the others are to be added in future versions. These include: Offset Pursuit, Arrival, Obstacle Avoidance, Wander, Wall Follow, Containment, Flow Field Following, Unaligned Collision Avoidance and Leader Following. Hopefully when these are

added the user will have a large repertoire of basic movement thoughts that can be combined into complex higher-level thought clusters.

SeekBot:

This module adds an aiming force towards a given bot's location. A t (time) value can be provided, so that the bot's position is predicted that far into the future. There is also a failure distance, which makes the thought a failure if the target is outside of this distance. As the methods are almost identical, there is also a flee attribute, which when turned on applies a force away from the target bot. When fleeing the failure distance is used to calculate when the thought is a success i.e. when it has got away from the bot that was chasing it.

SeekPoint:

This is almost the same as SeekBot, except aiming towards a fixed point rather than a moving bot. A possible addition to the SeekPoint module is to include a SeekVector module, which takes in a pointer to a Vector, and seeks that point. This way the point does not have to be stationary, and could represent any moving point in the universe.

The following three modules, Separation, Cohesion and Alignment, combine to produce the flocking behaviour initially conceived of by [REYNOLDS 87]. Flocking is an instinctual thought, and as such generally given a priority of 0, so it is always considered.

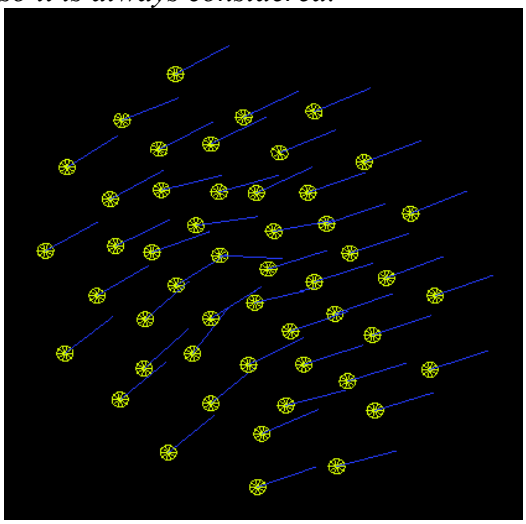

Figure 9 - A flock of bots in the same group demonstrating Separation, Cohesion and Alignment thoughts

Separation:

The Separation aspect of flocking means that a bot receives a force away from all of the bots in it's specified local neighbourhood. These repulsive forces are scaled by the distance of each bot, and summed together to produce an overall force. The Separation forces are taken from all bots, regardless of the group they are in. The main reason for this force is to keep the bots from colliding with each other (although it does not guarantee collision avoidance, and a future improvement would be the inclusion of a collision thought, which deals with a collision, bouncing the bot in a different direction).

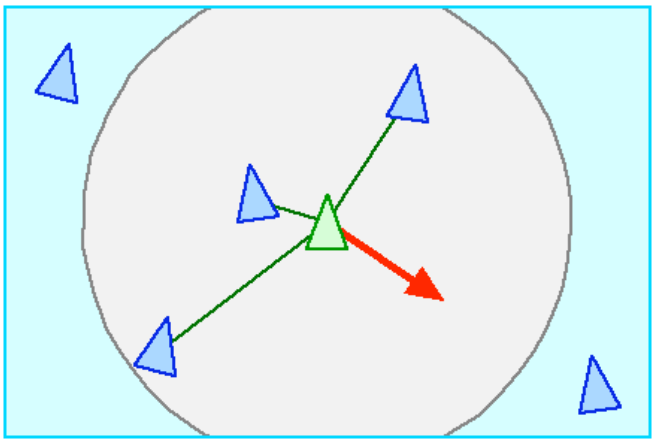

Figure 10 - Separation Module moving bot away from neighbours [REYNOLDS 99]

Cohesion:

The point of Cohesion is to group the bots together into gangs with neighbouring bots. The positions of every neighbour are averaged to give the "centre of gravity" and an aiming force towards this point is taken. Only bots that are in the same group are averaged, so that bots of the same group will clump together, ignoring other groups.

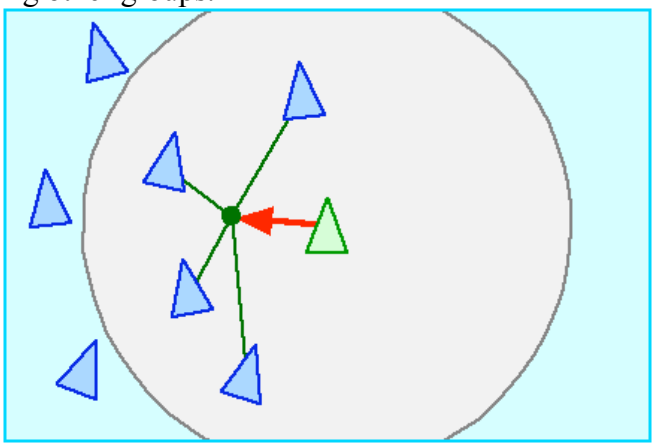

Figure 11 - Cohesion Module moving bot towards the centre of similar neighbours [REYNOLDS 99]

Alignment:

To complete the flocking trilogy is the Alignment module. This attempts to make the bot's direction and velocity match that of its neighbours from the same group. This gives the impression that the flock is moving together with purpose. It is interesting to watch the flock's behaviour when this module is included, as their turning abilities decrease dramatically, just like real flocks. It takes a group effort to turn rapidly, every bot slowly affecting its neighbour's direction, and vice-versa.

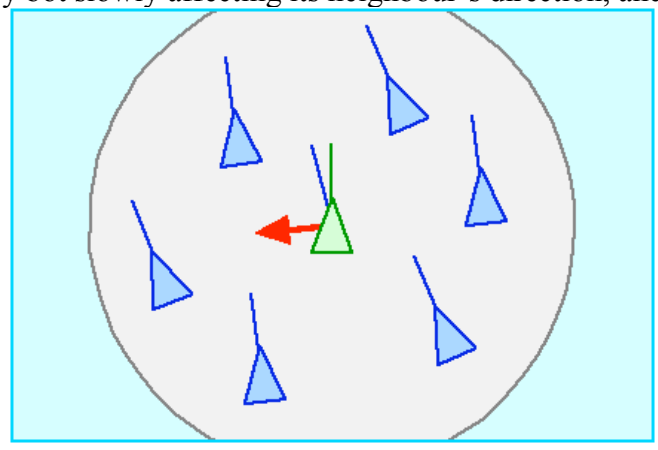

Figure 12 - Alignment Module aiming the bot in the same direction as its neighbours [REYNOLDS 99]

FollowPath:

The followPath module produced for this version is a rather simplified adaptation of the method described in [REYNOLDS 99]. It simply produces an aiming force towards a point on a specified path object (a poly line consisting of a list of coordinates). When the bot is within a certain radius of this point it changes its target to be the next point in the path. There is also an option to reverse the direction along the path it is travelling. In the very basic version used as an example in the program the direction is reversed when the bot reaches the end of the path. This way it keeps walking up and down the path. Although just a hack to keep the bot moving, this could be taken further and used as a patrolling thought, for a military style game.

RouteFinding:

This module was one of the originally proposed reasons for this project. However, as the scope of the problem became less specific and more about an overall AI control mechanism, the implementation of this thought process became less important. The final result is a little disappointing, and still requires a lot of improvement. It is not particularly efficient in its methods, but contains some ideas that can hopefully be used in later versions.

The route finding is performed on a specified map of data. The type of data in the map is what we use to calculate which route will be optimal. The algorithm used is A*, which is an advanced version of Dijkstra's algorithm.

> *"As we are only interested in the shortest path between two vertices, it seems like a waste to compute all the paths from one vertex to all the other vertices as Dijkstra's algorithm does. The most obvious improvement is to stop directly at the moment when the destination vertex…is retired….The A* algorithm, introduced in [NILS82], takes care of this by adding a heuristic in order to bias the search toward the destination and thus hopefully finish faster."* [JÖNSSON 97]

This heuristic function estimates the cost of travelling between two points on a map. There is also a function that is used to calculate the cost of travelling between two adjacent squares on the map. In the present version of the program these functions are part of the routeFinding thought, and as such do not successfully take into account the type of data the map represents, only viewing the maps as representing the maximum speed that the bot can travel at. An improvement that would solve this would be to keep the heuristic and cost functions in the map itself, and have different versions of these functions depending on the type of data the map contains.

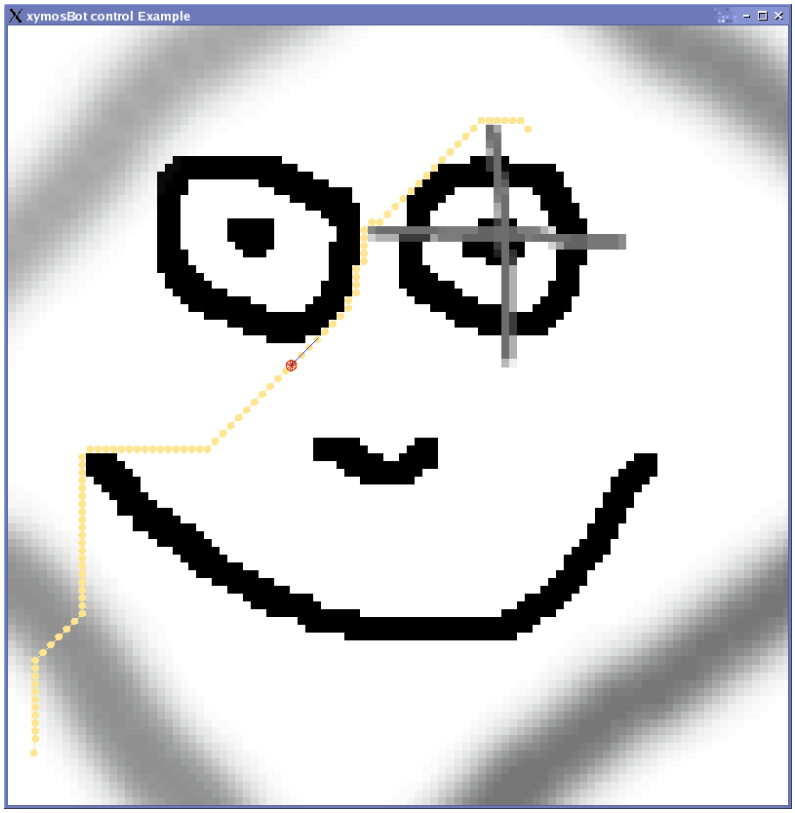

Figure 13 - A bot travelling along a route calculated with the routeFinding thought module

An exact description of the A* algorithm will not be described here, as there are very good explanations of it, and the many ways it can be optimized, in [JÖNSSON 97] and at [HEYES-JONES 04]. In future versions of this module a lot of these optimizations and techniques will be added, hopefully making this a much less CPU hungry thought module.

One step towards optimization that has been implemented is the ability to choose the number of steps that the thought takes every clock cycle. As it may take several seconds to reach a solution we do not want the whole program to hang whilst a route is calculated, especially if this is performed for many bots. By specifying how many steps of the algorithm we take we can break up the calculation over several clock ticks. If the number of steps is low it will take much longer to reach a solution, but will not use up much of the processor's time. If, on the other hand, the number of steps is high a solution will be reached a lot quicker, but will keep the CPU a lot busier. Additionally, if a value of 0 is used the route is calculated all in one go, not something that should be used very often, but the ability to do this is useful for certain situations. The number of steps could in future versions be changed on the fly, so that a consistent use of the CPU is maintained, so when there is free processor power it can be given to the route finding algorithm.

Improvements

Improvements that could be made to the program's current priority system include methods that automatically shift all the thought priorities up or down a level. For example, a quick method for inserting a new thought with a higher priority than all the other thoughts, and a method of reshuffling the list when a thought is removed would improve the simplicity of the system. Also, a non-priority that guarantees that a thought will not be performed would be very useful, as even giving a thought a massively low priority would not ensure it wasn't called.

In terms of the thought building blocks it would be useful to include a simple structure that allows for categorizing each thought into a library for later use. This would make for a more organized and user friendly approach to the AI system, giving the level designers less coding to perform. A possible addition (which is unlikely to occur, but would improve the usability significantly) would be a

graphical front end to the AI system. Something that would resemble Figure 6 and allow users to combine building blocks and set the criteria by which the thoughts are controlled. This, however, would be a very large task and would also remove the programmer from the game development, which isn't what we want.

There are additional types of thought building blocks that the author would be intrigued to implement in future versions. One idea is that of 'message thoughts', which broadcast a message to any bot that is within a certain radius. All bots have a 'listening thought', which deals with any message thoughts that are received. This way bots can choose to ignore messages or act appropriately. There could also be 'location message thoughts'. The universe would have a list of message thoughts, each one with location. When a bot nears this location it receives the message and it's listening thought handles the response. This is similar to the "smart environment" [ANDERSON 03] used in The Sims game, where every object is "annotated". This method could be used to provide very strong control over the actions of the bots. For example, if a plot device required AI characters to travel to a location, it would be possible to give the characters new instructions that furthered the plot when they reach that location, without them having to know in advance what they were going to do once they got there. Multiple and dynamic threads of story could be acted out based on the actions of the bots.

Another possible thought type is a 'bot thought', whereby a bot makes a copy of a target bot inside its brain, copying the bot's statistics and thought list (both the statistics and thought list will be limited to a specific set of possible perceptions i.e. only visible statistics and certain thoughts will be copied). It would then be possible to step through the target bot's thoughts and imagine its actions in the future. This could give bot's the appearance of high levels of intelligence and advanced predictive capabilities.

As this system is intended for use in computer games a way of controlling its drain on the CPU would be very useful. In the current implementation there is no camera in the world, but in future versions it would be interesting and useful to use the bot's location relative to the camera to alter the AI. This way, when a bot is off screen or far away its does not need to think as deeply about problems because it's actions are not being scrutinized. The bot's thoughts could be called every other clock tick, and actions such as fighting could be simplified to a statistics based outcome rather than calculating every hit and block. All of these possibilities would improve the performance drastically, something that is massively important with graphics being so intensive.

Conclusion

The result of this project has been to produce something of a complexity and scale that the author has never accomplished before in programming. Whilst initially it was intended to be a finished program that performed a single specific task the outcome has become something a lot larger and more open ended, and something that in the long run will be a lot more useful.

There are lots of issues with the current implementation of the program, many of which have been described in the preceding report. The user control of the program has been hacked together to at the last minute due to a lack of time. Also, the overall structure of the program does not completely tie in with the design of program's requirements. This is due to the author's desire to get down to the programming before fully planning the form of the program. Because of this there were several almost complete rewrites of the software before the current incarnation was produced.

Although nowhere near complete it is the concepts and structure of this program that are important, as they can now lead onto much more advanced and larger implementations. When time is available this AI engine will be advanced and used in several games that are currently being planned.

References

Anderson, E.F (2003) *Playing Smart – Artificial Intelligence in Computer Games*, in the proceedings of zfxCON03 Conference on Game Development, 2003

Funge, J.D (1999) *AI for Games and Animation: A Cognitive Modeling Approach [online]*
Available From: http://www.gamasutra.com/features/19991206/funge_pfv.htm
[Accessed 6 March 2005]

Heyes-Jones, J. 2004 - *Justin Heyes-Jones personal web pages A* tutorial [online]*
Available from: http://www.geocities.com/SiliconValley/Lakes/4929/astar.html
[Accessed 6 March 2005]

Jönsson, F.M (1997) *An optimal pathfinder for vehicles in real-world digital terrain maps* The Royal Institute of Science, School of Engineering Physics, Stockholm, Sweden

Nilsson, A.J (1982) *Principles of Artificial Intelligence*, Springer Verlag. Berlin.

Reynolds, C. W. (1987) *Flocks, Herds, and Schools: A Distributed Behavioral Model*, in Computer Graphics, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34.
Available from: http://www.red3d.com/cwr/papers/1987/boids.html
[Accessed 6 March 2005]

Reynolds, C. W. (1999) *Steering Behaviors For Autonomous Characters*, in the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California. Pages 763-782.
Available from: http://www.red3d.com/cwr/papers/1999/gdc99steer.html
[Accessed 6 March 2005]

Websites

GameAI.com - http://www.gameai.com/
[Accessed 6 March 2005]

NeHe Productions - http://nehe.gamedev.net/
[Accessed 6 March 2005]

OpenSteer Documentation - http://opensteer.sourceforge.net/doc.html
[Accessed 6 March 2005]

Cprogramming.com - http://www.cprogramming.com/tutorial.html
[Accessed 6 March 2005]

C/C++ reference - http://www.cppreference.com/index.html
[Accessed 6 March 2005]

MSDN Standard C++ Library Reference -
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcstdlib/html/vcoriHeaders_STL.asp
[Accessed 6 March 2005]

Programming With The Standard Template Library - http://www.robthebloke.org/
[Accessed 6 March 2005]

Table of Figures