AI Manual

2004-09-14

Index

Part I – General information
- Background information
  - Events
  - Rules
  - State

Part II – Shared systems
- Pathfinding & Navigation
  - General properties of the graph
  - Triangulation
  - Forbidden Areas
  - Link Data
  - A Small Triangulation Example
  - Pathfinding

Part III – Sensory models
- Vision
  - Visibility Determination details
  - Soft cover visibility and behavior
  - Perception coefficient and visibility

Part IV – Tactical systems
- Anchors and modifiers

Part V – Scripting
- Behavior Scripts
  - Setting up the behavior
  - Creating a simple behavior
  - Responding to an event in the behavior
- Goals and goal pipes
  - Goal pipes
  - Pushing goals in goal pipes
  - Selecting goals to execute
  - Goal pipe arguments and execution details
- Combining behavior scripts and goal pipes

Setting up a situation
Implementing reasonable responses
Character Scripts
The Character script concept
Making a new character
Relationship between behaviors and characters
Implementation and execution details
Behavior call indirection levels
Character indirection levels
General logic scripts
Idle animation manager
Conversations and conversation management
The name generator
Signals

Part VI – Tips and tricks
Cover objects
The (dreaded) "10 meter cover" rule
Simulating specific behavior with smart cover placement
Using soft cover in a smart way

Appendix A – Atomic goal reference

Appendix B – Lua script function reference
AI System script functions
Entity script functions
System script functions

Appendix C – Lua script callbacks reference

Appendix D – FAQ

Part ONE – General Information

In this part, the basic principles behind the AI system of the CryEngine are discussed. Conventions, background and rules are loosely discussed. This part will make reading the rest of this manual a more pleasant experience – since the system will be introduced to the reader on a very low level.

This part does not contain how-to instructions; it is more a theoretical overview.

Background Information

This section discusses some of the goals and principles that were set during designing of the AI system in Far Cry. Even though this was a long and iterative process, these goals outlined early on what was expected of the final product and how it would all work conceptually.

The primary goal was to define a system in which there are a set of rules and a framework in which new rules can be added. The ruleset needed to be valid for individual AI tactics as well as group tactics and it had to be simple and easy to grasp.

In order to understand the game rules of Far Cry we will start by making some definitions.

Events

Events describe everything that the player can perceive or can cause to happen in the game world. Examples of events are – shooting a gun, moving, rolling a barrel down a hill etc. When trying to derive the ruleset for Far Cry, it was necessary to organize all the events that can happen while playing the game (which are many) into some sort of hierarchy. In this way, hopefully we would end up with a small subset that we could then use to define our ruleset. Thus, it was necessary to differentiate between a *Game Event* and a *Tactical Event*.

Everything that can happen in the game world is a Game Event. This means that Game Events are the lowest in the hierarchy. The second level is, of course, the Tactical Events. Since Tactical Events cannot be defined as easily as Game events, we used the following rules when defining a tactical event:

- Different Game Events can generate the same Tactical Event;
- Multiple Game Events in a particular sequence can generate a Tactical Event;
- Combinations of Game Events and enemy properties can generate different Tactical Events

Tactical Events are the *same* for the player and the AI. The difference is that the player responds to a Tactical Event by pressing buttons on the keyboard and moving the mouse, while the AI responds by receiving directives from the AI system.

This "generic" quality of Tactical Events is why we have chosen them as the base for the ruleset that we will define.

Following is a table of Tactical Events and descriptions of each event as they are defined in Far Cry (Note that both player and AI are called soldier in the table):

Tactical event	Description
INTERESTING SOUND	The soldier heard a sound that poses no imminent threat to him. Examples: a footstep sound, a splash of water, a twig breaking etc…
THREATENING SOUND	The soldier heard a sound that threatens him. Examples: gunshots, explosions etc.
POTENTIAL TARGET	The soldier has acquired *some* target. He does not have enough information to determine whether it is hostile or not.
HOSTILE TARGET	The soldier perceives a REAL hostile target.
TARGET MEMORY	The soldier has lost his target, but has a memory of its last position
TARGET LOST	The soldier has lost his target AND any other helpers that could point to its position
GROUP COMMAND	The soldier issues or receives a group command (formation, alert level etc.)
DISTURBANCE	The soldier knows that somehow his environment changed, but he has no target to associate with it.

As can be seen from the table, there is not that many Tactical Events – but they sufficiently encompass all classes of Game Events that can happen during playing of the game. Just an example – there are numerous Game Events that correspond to the INTERESTING SOUND tactical event (footstep sound, a splash of water, a twig breaking etc); The target running behind a cover object, then sneakily escaping from the scene shows how a few Game Events in sequence can create a Tactical Event – TARGET LOST.

This table defines ALL the Tactical Events that the game Far Cry recognizes.

Having defined the base building blocks of the rules in the game, we can continue by defining the rules themselves.

Rules

The rules are defined on top of Tactical Events. These are the only two formats in which a rule can be defined:

- A rule should define what causes a Tactical event
- A rule should define a response to a Tactical event

Since in the previous section we have already hinted what can cause a Tactical event, a final set of rules would clearly list which Game Events (and combinations thereof) correspond to exactly which Tactical Event. Then we would create a final list of responses to those tactical events based on the characteristic of the AI object we want to simulate.

As an example of this process (as it is obvious that we cannot list the entire ruleset of every character personality in the Far Cry AI) we will take a detailed look in the rulesets of two enemy characters: the **Cover** and the **Scout**. These are the two most common human enemies in Far Cry and as such feature very detailed rulesets. More information about the actual implementation of these two characters can be found here.

To understand how we connect Game Events to Tactical Events, we will look more closely at the THREATENING SOUND event. Figuring out which Game Events fall under this category is done by common sense and by following examples of how a real soldier would react to some event. It is obvious that a gunshot sound would be a threatening sound since it rarely means something good, but on the other hand - the sound of a knife hitting a table is something vaguer. Under some circumstances it could be threatening but in some it could be completely harmless.

It is important to select the smallest possible subset of Game Events for a particular Tactical Event, **since the response of the enemies to the same Tactical Event will always be the same**. It would be silly for an enemy to get scared to death if his friend happens to drop the knife that he is using to prepare dinner on his table ☺

So, deciding on the exact subset of events that go under the THREATENING SOUND category has largely been a judgement call in Far Cry's design – that, and countless hours of adding and removing events and play testing for consistency. Eventually, it was decided that the events that fall into this category would be:

- All weapon shooting sounds
- All explosions

Now this may look like a small subset, but in fact it includes a lot of things that can happen in the game, for example detonation of projectile weapons, explosions of various destroyable objects and the shots of all the varieties of weapons in the game (and there are a lot of them).

So now, the rules extrapolated from the previous discussion can be written down in a table form:

Tactical event	**Game events**
THREATENING SOUND	Weapon shot
THREATENING SOUND	Any explosion
DISTURBANCE	Bullet impacts close to the soldier
DISTURBANCE	A friendly unit has died
…	…

The table also contains some examples for the DISTURBANCE tactical event, just to illustrate that the final rule table should contain all possible tactical events and everything that is set to generate that tactical event. A complete list of tactical events and which game events correspond to them is given in Appendix ??.

All that is left once we have all the rules that map the Game Events to Tactical Events is to write out the rules that govern the response of an enemy to a tactical event. Obviously, the responses of different AI characters would be different, so we would need to create separate rules for separate characters. Indeed this is what has been done in Far Cry.

In the example for the Cover and Scout characters, we will create the tables that describe the responses of the individual characters to a specific Tactical Event. Let's start with a small overview of the major character properties:

Cover: Has small to medium attack range; Uses rapid fire weapon; Always fights to pin down his target (uses cover to advance); Always tries to minimize distance between himself and target; Very responsive to group signals; Low general alertness (he is easy to circumvent, but very hard to lose after he perceives the target); He is rather slow moving.

Scout: High general alertness (very nervous, thoroughly investigates every little disturbance, hard to circumvent, hard to shake off once he sees the target); Has medium attack range; Uses a powerful weapon (rapid fire or pump-action); Medium responsive to group signals (he disregards some signals from the leader); Always tries to seek out an alternative approach to his target (flanks a lot using cover); Keeps medium distance from his target; He is fast moving, light on his feet.

Starting from these general properties (that represent somewhat of a "wish list" of what the characters should do), we can create the rule table and carefully select a response appropriate to whatever the enemy has perceived. Later that rule table will be effectively translated into a working script and an actual enemy present in the game itself. The tables of responses that are implemented in the game are summarized here:

Tactical event	COVER RESPONSE
INTERESTING SOUND	The soldier draws his gun. He starts slowly approaching the source of the sound on the shortest path. He does not alert his team yet – it might be nothing.
THREATENING SOUND	The soldier draws his gun while making a scared animation. He first searches for the nearest cover to his position. After he reaches the cover he comes out carefully while trying to pinpoint the source. He alerts his team
POTENTIAL TARGET	The soldier draws his gun and yells a warning message. He tries to get closer to the target via the shortest path. He does not alert his team.
HOSTILE TARGET	The soldier starts shooting while going for the nearest cover. He notifies his team.
TARGET MEMORY	The soldier tries to advance forward by running to the next cover object in the direction of his target. If no cover objects then runs to his target via the shortest path.
TARGET LOST	The soldier stops pursuing the target. If he did not see a hostile target or hear a threatening sound, he goes back into idle state. Otherwise he remains alerted.
GROUP COMMAND	**These depend on the actual command**
DISTURBANCE	The soldier draws his gun, and tries to approach a cover object, where he crouches and tries to spot a target by looking around.

Tactical event	SCOUT RESPONSE
INTERESTING SOUND	The soldier draws his gun. He investigates the sound by approaching him while flanking left or right. He does not alert his team yet – it might be nothing. Drops an invisible marker at the sound position.
THREATENING SOUND	The soldier draws his gun while making a scared animation. He first searches for the nearest cover to his position. After he reaches the cover he comes out carefully while trying to pinpoint the source. He alerts his team
POTENTIAL TARGET	The soldier draws his gun and yells a warning message. He hides and tries to approach the target by moving around him. He does not alert his team. Drops an invisible marker at the target position.
HOSTILE TARGET	The soldier starts shooting while going for the nearest cover. He notifies his team. If target too close, runs back to a safe distance.
TARGET MEMORY	The soldier tries to flank his target by selecting cover objects to the left or right of him. Effectively trying to circumvent the target position. Drops an invisible marker at the target position.
TARGET LOST	The soldier now relies on any invisible markers he dropped previously while in combat. He carefully circumvents and tries to approach the markers, looking to reestablish contact with the target.
GROUP COMMAND	**These depend on the actual command**
DISTURBANCE	The soldier draws his gun, and tries to approach a cover object, where he crouches and tries to spot a target by looking around.

While in some respects these two soldiers react the same or similar to each other, in most respects they are somewhat different. This forms the whole basis of the systematic approach to enemy design that Far Cry has chosen. All covers and all scouts will react like this given a particular Tactical Event, regardless of positioning. The individual encounters can be choreographed further by the specific layout of the environment.

Also, these tables only list the basic functionality of the given enemy types. This would indeed make for a very repetitive and dry experience, so this system had to be extended with certain modifiers of the original response model – in the shape of hint objects that are left in the environment (called Anchors). Enemies then fish for these hint objects in their immediate surroundings and modify their basic behaviour in accordance with what they find. This is discussed in more detail in this section.

Another thing to notice is that the responses listed in the table are largely valid ONLY if the enemy was previously in an idle state. This gets us to the final theoretical construct in the Far Cry ruleset, and that is:

State

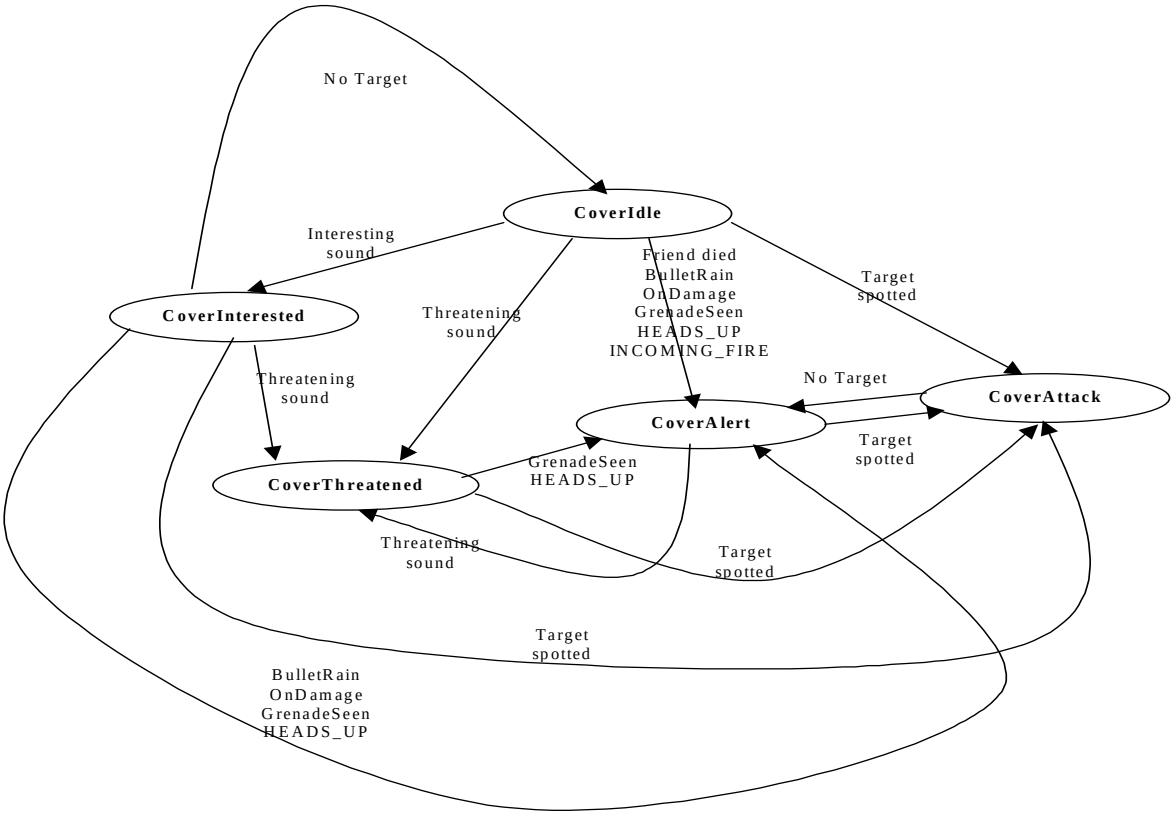
The state of the enemy describes the current state of emotional awareness. Far Cry does NOT predefine states, and every enemy character can define as many or as little states as it needs. At the very least, there is the idle state that describes the character before he had any targets. Some of the other more common states are Attack, Alert etc.

It is obvious that just a single ruleset cannot possibly cover all aspects of the character's response. As an illustration, a character would respond to a threatening sound differently if he were in an idle state – and differently when he is already alerted, or even actively engaged in combat. While a threatening sound carries significance in idle mode, during combat it is not so important, because the enemy "understands" that he is in combat, and gunfire and explosions are normal occurrences in combat.

Certain events will have to take the character from one state into another. This is obvious as an enemy who has heard an explosion is not idle anymore; instead he needs to move into alert state. The rules on how to go from one state to another are the last thing we need to clearly outline to be able to completely define a character.

Thus to complete the whole description of a single enemy character – we need to outline ALL the states that this character can find himself in, and then list out the entire response rulesets for all states for each individual character. This has been done in Far Cry, and then it has been effectively translated to script code. All the theoretical constructs discussed here (rules, rulesets, states and state transitions) have appropriate representations in the script. This is however discussed a little later.

Here is an example of the state diagram for one of the enemies (the Cover), which outlines the individual states and the rules that define how to go from one state into another.



Further in this document, it will be explained how to create new characters for the Far Cry game, as well as how to analyze existing ones. This section aim was to introduce the principal way of thinking behind the design of the enemies in Far Cry and provide some more insight into ways to plan out any characters that might be considered for inclusion into the Far Cry base code.

PART TWO – Shared systems

This part discusses in detail some of the subsystems of the AI system each of which specializes in performing a specific task. This includes a description of how Far Cry's environments are represented in the AI system and how navigation and pathfinding is executed within that representation. Several peculiarities about the functionality of the Far Cry AI system and certain optimizations will be presented here.

People who want to design levels for AI characters in Far Cry have to read this part of the manual as it will describe how the most important aspects of the system work and common problems and pitfalls in developing levels for Far Cry.

Pathfinding & Navigation

The CryAISystem uses a non-directed graph as the basis of its spatial representation. Parts of this graph are calculated in a pre-process step in the Cry Engine Sandbox ™, and some are processed in real time as the game executes. The properties of this graph and how it is used to generate paths and navigation is discussed in this section.

General properties of the graph

A graph is a data structure that consists of two data types – nodes and links. The nodes represent the repositories of the data organized in the graph, while the links provide connections between the nodes (and thus between the data contained within the nodes). A non-directed graph means that there are no directions associated to the links (or that every link is bidirectional) – in essence when one node is connected to another that automatically implies that the other node is also connected to the first.

In the Far Cry AI system, the graph nodes represent a certain portion of space that is clear of obstacles. The data they contain describes this portion of space using some properties. The links contain data that is used to calculate whether a certain agent can pass from one node into another – so they are not just simple links but they also contain data – like the nodes. There are no limits to the amount of links a node has.

Nodes can be created in two ways – there is the automatic generation of nodes using a triangulation procedure (obstacles are triangulated on the map), which generates about 90% of the graph. The user does not directly affect this operation and it can be initiated from within the Cry Engine Sandbox ™; this is a pre-process step. The second way to create nodes in the graph is by manually placing them in the Cry Engine Sandbox™ and then manually linking them together to the underlying automatic generated graph.

The auto generated graph ALWAYS HAS TO EXIST, even if the map is completely empty. Within the CryAISystem, the graph is never 0 - it always contains at least one node that describes the whole game world.

Triangulation

The generated triangulation is the core of the navigation algorithm. It is performed in order to calculate the initial graph, which can then be modified by the user or used as is. The basic principle behind the triangulation is that it creates a triangle between each three obstacles that are in some proximity to each other. The resulting triangle consists always of free space that the agents can move through, because the vertices of the triangle are the obstacles themselves. The edges of the triangles govern where the agents can pass from one triangle into the adjacent one. The triangles then become nodes of the graph and the edges become links between the triangles.

Why was triangulation chosen? Even though it provides a reasonable approximation to the actual game space, it takes up the smallest amount of memory since it packs vast open spaces without obstacles in a single triangle. It does not require that the designer of the level spend a lot of time placing manual waypoints (although this is still a possibility for designers that really want to control the environment). Finally, the triangulation topology lends translates very elegantly into a graph, and the operation and the way paths are calculated are easy to grasp within a triangulation (hopefully).

Following is a detailed description of all the sub steps that are performed in order to generate a final graph. Every step will be accompanied with an example that will try to explain the procedures in a more visual way.

When an empty level is created, its triangulation is non-existent. However, the graph still contains one dummy node that represents the whole empty level. If a triangulation is generated on an empty map, it will create dummy obstacles on the edges of the map (in total 4) and will generate 2 big triangles that connect these 4 dummy obstacles. Thus the graph will contain a total of 3 nodes – the one dummy node that is always created – plus the two nodes each of which describes one of the triangles. The two triangle nodes would have one link each to the other, while one of them (the one that is created first) will also contain a link to the initial dummy node.

When a triangulation is initiated on a normal level (one with obstacles), the first step is to request from the physics subsystem ALL physical entities that collide with the player (e.g. obstacles). All the obstacles returned are taken as simple points (without volume). The point position taken for the triangulation is the origin point in the objects local space. Note that in this step the size of the actual obstacle or its shape is not relevant – so big buildings and small rocks are all handled in the same way – as a point.

This set of points received from the previous step is put through a triangulation process that generates a triangulation of the points in 2D. This triangulation is something similar to a Delaunay triangulation, although it is not quite the same. The Delaunay triangulation has some more strict rules about the properties of the triangles that come out as a result of the triangulation. A demonstration of the process of the triangulation can be found in many places on the Internet (http://cage.ugent.be/~dc/alhtml/Delaunay.html). This is performed offline

(as a pre-processing step) as it can be potentially very time consuming (depending on the amount of obstacles and their distribution).

The triangulation generated in the previous step is used to create the navigational graph. Each triangle is translated into a node (its geometrical centre is taken as the position of the node) and each triangle node is connected to the nodes of all its adjacent triangles. Basically, these are all the triangles (actually exactly 3) that share an edge with the node's triangle. The nodes also record properties of the triangles at this point that may or may not be used during pathfinding (does the triangle border a water area, the slope of the triangle etc). With this procedure an early version of the navigational graph is created with the following properties:

- Each node has exactly 3 links (except one node that has been created initially which also links to the dummy initial node that is always created)
- The links describe a sort of spatial relationship between the triangles – in other words they can now easily determine which triangle is neighbouring which.
- The triangulation itself represents a 3D plane that follows somewhat the surface of the terrain (how closely depends on the obstacle distribution)

At the end of the previous step we have a working graph that represents some kind of spatial information, but it's incomplete as it regards all obstacles as points. Also there has been no influence from the game designer thus far to put some areas of the map out of reach for the enemies. To remedy this situation, the designer can place Forbidden Areas in the Cry Engine Sandbox™, which are processed next.

Forbidden Areas

In every game there are areas where the designer would like to keep the enemies out. Examples for this are water surfaces, steep hills, ravines etc. To make it possible to indicate to the AI enemies in Far Cry that they should not go into some certain areas the designer can use Forbidden Areas.

The Forbidden Areas are 2D shapes that consist of three-dimensional points. What that means is that while the forbidden area doesn't necessarily need to be flat, the difference in the z coordinate for the points is irrelevant – the closes analogy is that the areas are infinitely high prisms. **The edges of the forbidden areas define infinitely high walls that the enemies can never cross**. The area enclosed by the forbidden area is NOT actually forbidden. An enemy can freely be spawned there and he will work normally. BUT, no one from outside of the forbidden area will be allowed in, and no one from inside of the forbidden area will be allowed out.

In this sense, it would be better to call these areas Forbidden Shapes to indicate that it is indeed the edges of the shape that are forbidden to cross – but the name is historic. An enemy inside a forbidden area will not make an effort to exit it.

If some enemy tries to generate a path whose destination would take him outside of the forbidden area that is enclosing the enemy, the path will be quickly rejected as impossible. The same will happen if an enemy tries to generate a path whose endpoint is within some forbidden area that doesn't include the enemy as well. It has to be said however that enclosing enemies inside forbidden areas is not a

good idea as their primary purpose is to enclose portions of the map where the designer does NOT want the enemies to be.

Edges of forbidden areas are included into the triangulation graph after it has been created. As edges are added to the triangulation, triangles are split accordingly making it so that every forbidden area edge always coincides with the edges of one or more triangles. The links created that correspond to these forbidden edges are specially marked as forbidden (not passable). As a result, the triangulation generated in the first step gets fragmented to smaller triangles along the forbidden areas edges. The graph is kept in a stable state after each edge is added, so it is possible to have non-closed forbidden areas. Unfortunately in the present code state of Far Cry this is not supported – all forbidden areas are assumed closed automatically.

Link Data

Finally, the last step of the automatic triangulation is to analyze the edges of every triangle and determine which part (if any) of the edge length represents empty (passable) space, and which part is not empty (but inside an obstacle). This is the step in which the actual volumes of obstacles are taken into consideration, and depending on the number of incident triangles to an obstacle vertex in the triangulation a more or less accurate approximation of the obstacle's volume is created. At this stage, all obstacles are abstracted as infinitely high prisms and only the space between them along a triangle edge is marked as passable.

When an enemy generates a path through the graph, it also analyzes the links of the nodes as it traverses them and makes sure that it can "fit" in the empty area of the edge between the two triangle nodes before it takes the next node into consideration. Edges that have been created as a result of the addition of the forbidden areas are automatically marked with a negative amount of free space so that implicitly means they can never be crossed.

The analysis of the edges is done by shooting a beam (fat ray) from the first vertex on a triangle edge towards the second and recording the point the beam hit the obstacle that is represented with the second vertex. Then the same is done from the second vertex towards the first vertex obstacle. The two collision points are compared and the distance between them calculated – which can be negative if the two obstacles overlap. If the distance is positive, then it represents the maximum value for the diameter of an upright cylinder that can pass from one triangle into another through this edge.

The information about the diameter is stored in the link itself, and that means that in some way the links are weighted from the start based on the edges that they represent.

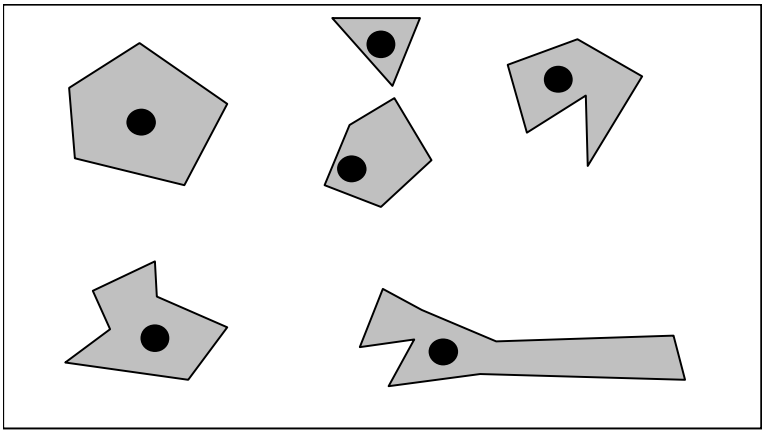
This is the last step before the finalized graph is written into a file. These files are with the extension .bai (binary AI) and they are loaded as-is in the game itself, so it is responsibility of the designer of the level to keep his bai files up to date.

A small triangulation example

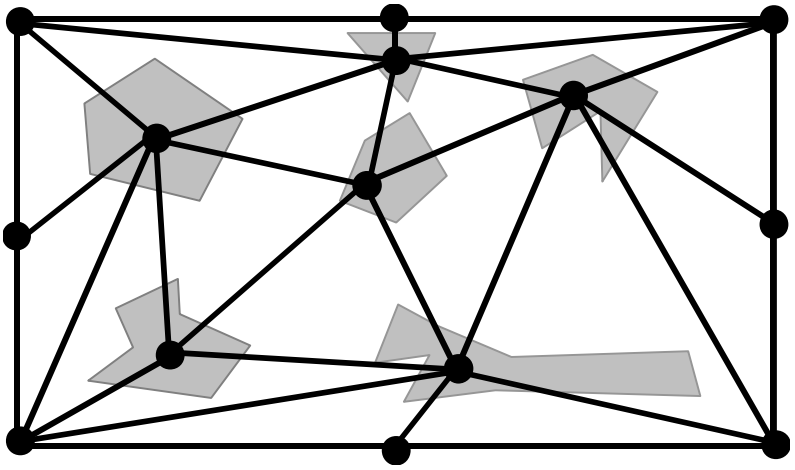
Because long drawn out discussions about what does what in a computer program can be too much to read, here is a simple example that illustrates how the

triangulation works and what is done at what step. It is important to understand the triangulation in order to understand how to make the enemies behave in the most believable way. The example is given in 2D because essentially the triangulation operates on a 2D plane, which is the terrain plane. How to extend it to 3D is discussed later in this chapter.

Our example setup looks like this obstacle course represented in the following figure. The obstacles are the grey polygons, and the origins of their local spaces are represented with a black dot. The forbidden areas in the picture are shown in red colour. We will follow the complete triangulation of this simple setup step by step.

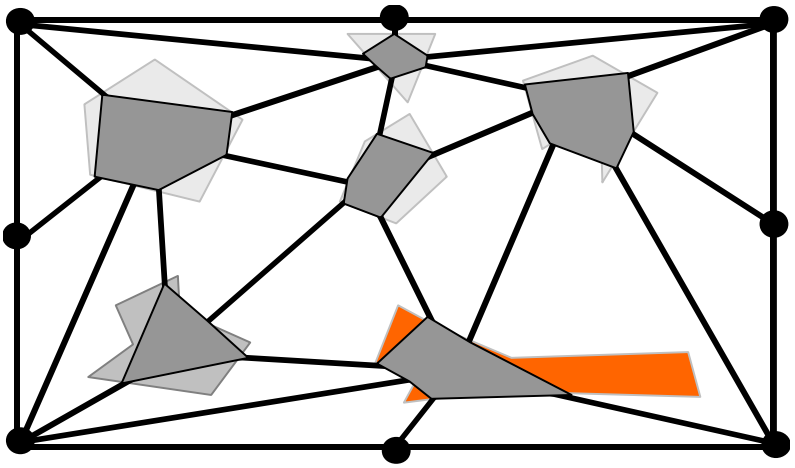


As discussed above, in the first step all obstacles are enumerated from the physics sub system and their positions (the black dots on the picture) are taken. These are then triangulated into triangles to receive the result in the following picture (Note that dummy objects have been placed on the edges of the world to enclose it):

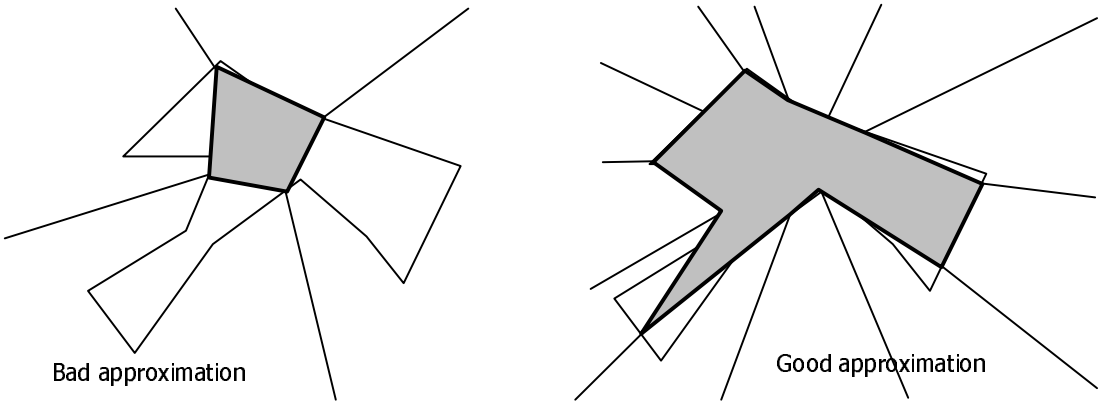


The obstacle volumes are greyed in order to show that for this stage they are not relevant. To illustrate where forbidden areas are indeed needed, we will skip the step that adds the forbidden areas (since there are none yet) and go ahead and calculate the pass properties of each triangle edge – to get an idea of how we

approximate the volumes of the obstacles. We would get something that looks like this:



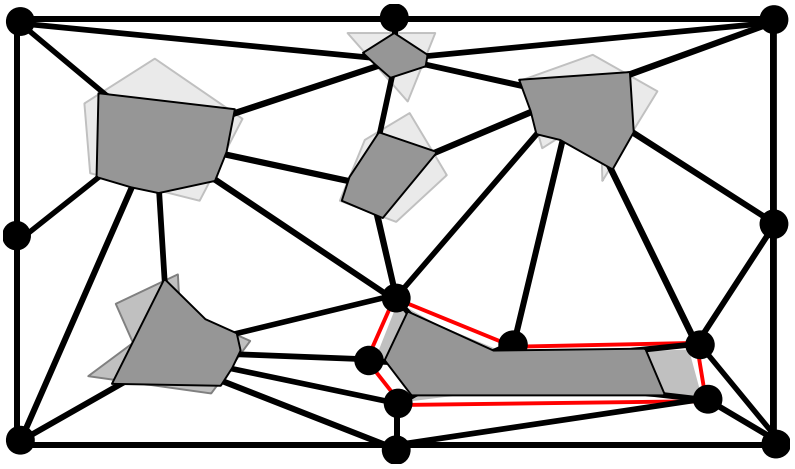
We can see from this picture that not all shapes are approximated in a satisfactory way (especially shapes elongated in one dimension). An example of a particularly bad approximation is the red polygonal obstacle – we see that if we regard all triangles as areas of empty space (without obstacles) we would be wrong in at least one triangle. For maps with a big amount of obstacles (like a typical Far Cry map that has a lot of vegetation) the number of incident triangles per vertex is bigger, and the obstacle volumes are more convex – so usually the approximation is good enough. In general though, the approximation is only as good as the amount of incident triangles – more triangles, better approximation. This is illustrated in the following image where the same obstacle is shown to be approximated good and bad.



Here is where we can make manual adjustments by defining a forbidden area around the obstacle that approximates bad (with experience, one can usually tell which shape would be problematic by just looking at the obstacle – buildings and all BIG objects are always prime candidates☺). The forbidden area provides a simple and controlled way to locally influence the triangulation and make it possible for it to generate a better approximation of the environment. As will be discussed later, during generating of the path the agents "think" that the obstacles are a little bigger (it's a sort of Minkowski sum of the approximated polygon and a sphere that depends

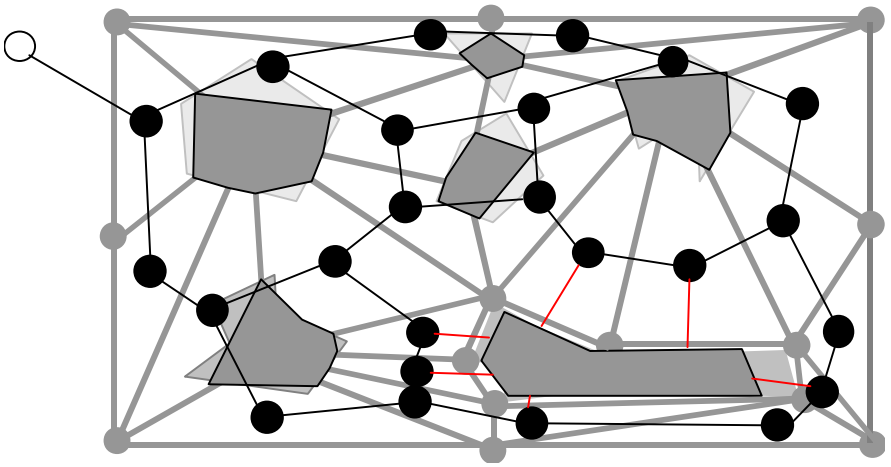
on the size of the agent and how it is configured in regard to how close to the obstacles he wants to pass) but still for obstacles that are really poorly approximated one needs to define a forbidden area.

All this being said: if we define a forbidden area around the obstacle shown in red in the diagram and we get the following result.



Notice that the forbidden area also contains triangulation inside of it (the approximated shape of the problematic obstacle is made transparent for that purpose). The problematic obstacle is now ideally approximated (by the forbidden area) and no agent will be allowed to step inside it. The addition of the forbidden area also added more points to the triangulation (all vertices of the forbidden shape) and this also refined the approximations of the surrounding obstacles, since now there are more incident triangles (for example the bottom leftmost obstacle and the top leftmost obstacle).

To see how this calculated spatial representation translates into a graph that is stored in memory and then used to pathfind and navigate, the following diagram superimposes the graph on top of the result of the triangulation.



The graph contains one additional node (marked as white on the picture). This is the so called *safe node*. It is always valid and cannot be deleted by anyone but the graph itself and is supposed to be used to always have an entry point into the graph that is guaranteed to be connected to the rest of the graph. Since there is no root node for the graph, the last returned node from a graph query is stored as such. That node is called the *current* node. Since sometimes it might point to a deleted node or even to null, the safe node provides a guaranteed way to get back into a valid portion of the graph.

The final observation to be made is that the nodes of the graph correspond to triangles in the triangulation, and the links of the graph correspond to triangle edges. The links themselves hold information about the edge that they describe (how big it is, what portion of it is free space etc). This information is critical during pathfinding and path beautification.

Pathfinding

The pathfinding algorithm uses a standard A* algorithm (http://encyclopedia.thefreedictionary.com/A-star%20algorithm) implementation to generate the path. It works on top of the spatial representation described in the previous text.

A path can be impossible if the two end nodes are on opposite sides of a forbidden area. Links that lead over forbidden area edges are not passable and are marked as having infinite cost (in the picture of the example triangulation they are coloured red).

During the A* pathfinding process, care is taken to make sure that the object requesting the path can indeed pass a certain link to another node before that node is considered. To this end, links have a lot of properties that aid during pathfinding and make it easy to calculate whether a particular path requester can "fit" through the edge.

When a path is found, it is often not of a high enough quality to be taken as is for the AI object to traverse. This is a natural side effect of triangulation, as paths can be sometimes very "broken up", with fairly large turns between points. To remedy this, a second pass is done on the generated path in order to straighten it as much as possible to facilitate more realistic movement of AI Objects when they traverse it. This operation is called path "beautification".

PART THREE – Sensory models

This part describes the modelling and principal operation of the sensors implemented in the CryAISystem, namely the visual sensors, sound sensors and a general purpose signalling mechanism that is included here for completeness even though it cannot be called a sensor in the strictest of terms.

Processing the sensory information is done on a full update of each enemy, even though the actual time at which a sensory event was received is asynchronous. The sensors are the only interface the enemy has with the outside world, and they provide data from which the enemy then assesses his situation and selects potential targets. All sensors are completely configurable in all aspects that they offer, and they can be turned on/off at runtime for any individual enemy.

Vision

The visual sensory model is the heart of the CryAISystem. It provides the enemies with the most important sense they have and it tries to simulate it as realistically as possible while still maintaining a low execution cost. To be able to meet these two often-opposing goals, various compromises and optimizations had to be done, but the end result is a quite satisfactory simulation of vision.

Every full update for each individual enemy the system traverses all potential targets from the enemy's point of view and puts them through a visibility determination routine (which is discussed in detail later). All targets that survive this filtering procedure are placed in a visibility list that is maintained until the next full update. For a target to persist as visible it has to pass the visibility test for every subsequent full update – otherwise it will be moved into the *memory targets* list which means that the enemy will now remember that he has seen this particular target before, but its not seeing it anymore. If some target is not visible for a few updates, but becomes visible again, it is removed from the memory target list and then goes back into the visibility list. The memory targets have an expiration time (basically a time interval it takes the enemy to "forget" about a target). This is defined by the threat index of the target, the time it was "exposed" – visible – as well as some other factors. Visibility targets have the highest priority and will be taken as the current attention target even if there is another target with a higher threat index. This is done in order to simulate the natural tendency of humans to act faster based on what they see rather then on what they remember, or hear.

Visibility determination details

Here the actual visibility determination procedure will be dissected and explained. First and foremost we will define a few terms that we will use in the explanation. The **sight range** of an enemy is a floating-point value that determines how far (in meters) a particular enemy can see. The **FOV** (field of view) of an enemy is a floating-point value that determines how wide the visibility cone of the enemy is (in degrees). The visibility cone has its tip at the enemy's head and extends outwards in the forward direction (out of the enemy's face). The **perception**

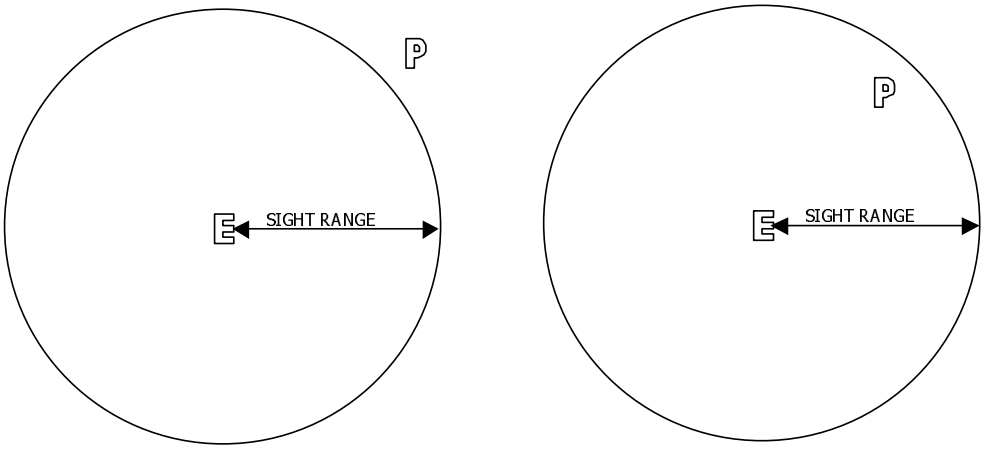
coefficient (SOM value) is a floating-point value in the range between 0 and 10 that defines how close the enemy is to actually seeing his target. An explanation of how the SOM coefficient is used is given here.

Before a potential target is EVEN CONSIDERED to be run through the visibility determination routine it must be activated – so the initial check is if this target is even currently active. If not, it is rejected.

Furthermore, since visibility determination can be potentially very CPU intensive, there is no visibility checks performed for enemies of the same species as the one that is currently being processed. Within the simulation, this is rationalized as follows: For example the mercenaries in Far Cry do not perform visibility determination among themselves because they would potentially already know where their mates are and what their strategy of combat would be. **Visibility determination is done only between different species**, for example between mercenaries and mutants. A discussion on species can be found later.

There is a mechanism to register ANY AI object as an object that should be included in the visibility determination (even user defined objects). This is done for the grenades in Far Cry, as well as for the flashlights etc. There are even special objects called attributes, which will be discussed in more detail later in this section.

The sight range defines a sphere with its centre coinciding with the position of the enemy's AI object. When determining whether a certain target is visible, first a quick and dirty check is performed to see whether this target is even in the visibility sphere of the given enemy. This is obviously done by comparing the distance to the target to the sphere radius. This is illustrated on the following figure:

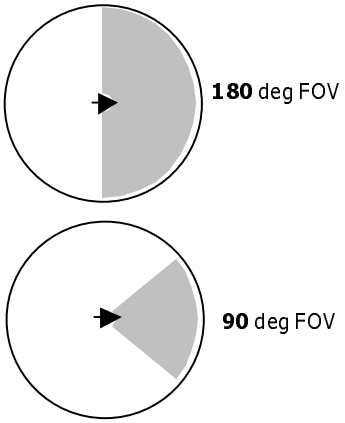


A. Player (P) is outside of enemy sight range, so he is quickly rejected as possible target.

B. Player (P) is inside of enemy sight range, so he passes the initial filter.

Targets outside the sight range sphere are culled early since the check is cheap. The next step is to determine which targets (that pass the sphere check) fall within the FOV of the enemy's AI object. The FOV is the angle that determines how far to the left and to the right of the current forward direction the enemy can see. The value that is specified for the FOV is not the half-value – meaning it is the angle between the two border orientations that limit the visibility. For example, a FOV of 180 degrees means that the enemy can see everything which is less than 90 degrees away from his forward orientation, while a 90 degree FOV means his ca see less than 45 degrees. This is illustrated in the following illustration.

Only those targets that fall inside the FOV cone (displayed by a greyed area in the picture) will continue the process of visibility determination. The rest will be culled out at this stage. The check is performed by a simple dot product between the orientation vector of the enemy and the vector created as the difference between the position of the potential target and the position of the enemy. The resulting scalar is then compared to the value of the FOV. **Note**: Although the illustrations are 2D the check is performed on a conical volume in 3D with its tip at the enemy position and its base perpendicular to the orientation vector of the enemy.



The targets that survive these two initial quick checks are very likely to be seen. The final check that is performed is an actual ray trace through the game world- and as such is the most expensive. Since the low layer of the AI system performs distributed updates over all frames it is very seldom that a large amount of rays need to be shot per frame. The only exceptions are scenes where there are indeed A LOT of enemies that belong to different species and huge combat scenes (with an excess of 20 participants per species). Note that this is not the case when the player faces a single species – for example humans – since they do not perform visibility checks within the same species, so effectively visibility determination is done only against the player.

The visibility physical ray is used to determine whether there are any physical obstacles between the enemy and his target. It originates from the *head bone* of the enemy character (or if the enemy does not have an animated character, it originates from the entity position – which is often on the ground), and it is traced to the head bone of the target (if it has one, otherwise the entity position is used). If this visibility ray hits anything in its path, then the target is rejected as not visible. On the other hand, if the ray reaches the target, then it is considered that the target has passed ALL TESTS and can now be added to the visibility list for this update.

An additional property of this last test is the TYPE of surface the visibility ray hits. In essence, there are generally 2 types of surfaces that the AI system discriminates – the so-called *soft* cover and *hard* cover. The primary difference in a physical sense between soft and hard cover is that the players of the game can pass through soft cover, while a hard cover object is an actual obstacle. In Far Cry, players can also hide behind soft cover objects but the visibility determination is a little "skewed" when the target is behind a soft cover object rather than behind a hard cover object or just in the open. So skewed, that it merits its own note.

Soft cover visibility and behaviour

One additional factor comes into play when determining visibility behind soft cover, and this is whether the enemy for which we are determining visibility already has a living target (living meaning not a memory, sound or other type of target). If the enemy does NOT have a living target, then for all intents and purposes soft cover is equal to hard cover and normal visibility determination is performed. This happens for example when the enemy is idle, or when the enemy is looking for the source of a sound, but has not yet spotted it.

However, when the enemy already has a target, then things change a little bit. If during the last stage of the visibility determination routine (the ray shoot) the enemy detects that there is only soft cover between him and his target, then he will CONTINUE to SEE the target further – for some time. This time is between 3 and 5 seconds. If within this time period, the target remains behind soft cover, the enemy will eventually lose the target and place a memory target at the last known position. If however the target comes out from behind soft cover within this time, then the timer is reset and normal visibility rules will come in effect.

The rationale behind this feature is that when a soldier perceives that his target runs inside a bush (for example) he does not immediately lose contact with it, since he can make out the target's silhouette even inside the bush. But following a target like that is hard over time, and after a while the soldier will lose the target. The same rules apply for example for wooden cover, but the explanation of the rationale behind it is a little different. In the case of the target running behind a thin wooden wall, the soldier knows that his bullets will still pierce the wall, so because he thinks he still sees his target, he continues to shoot through it. This makes for some really intense situations in Far Cry.

Of course, in order for all this to work in a closed and rational system, all surfaces in the game should be properly physicalized (wood, grass, glass should be soft cover, while rock, concrete, metal should be hard cover). This is consistently done in Far Cry.

This visibility test is performed automatically for all enemies of different species, the player and any other types of AI object that the users of the system want to add. Even though it is customary (and normal) to assign a special species to the player different from that of the AI enemies, the visibility check for the player and user defined types will be performed anyway – even if the species is the same. This is done to prevent unnecessary confusion and to ensure that there is a way to accurately specify the player as an object type that is always taken into account when checking visibility.

The AI System (at the lowest level) maintains a list of object types that should be included in the visibility check. Objects can be freely added and removed to this list, even from script. All that is needed is to specify an assessment multiplier to the desired object type and it will be included in the list. For an example of how this is done in the game Far Cry, refer to the file "aiconfig.lua" which can be found in the scripts folder. More about the nature of the multiplier that is being set with this function can be found in the threat assessment chapter.

The final detail that needs discussion and is related to the visibility determination is the so-called ATTRIBUTE object. An attribute object is not a full AI object in its own rite, it is more of a special helper that can be attributed to an existing AI object and can serve as a helper while performing visibility determination. The attribute is a special class of an AI object specifically defined at the lowest level in the AI system. Every attribute object must have a *principal* object associated with it. The principal object can be ANY type of an object (including puppet, vehicle, player etc) but not an attribute. When an enemy determines that he sees an attribute object, the system will actually switch the attribute with the principal object before adding it into the visibility list of the enemy. Thus, the enemy who saw the attribute will indeed think that he is seeing the principal object attached to this attribute.

Now you are surely thinking – what possible use can this "exotic" feature offer? Well, it is just a systematic way of connecting certain events to a single target. For example in Far Cry: When the player switches on the flashlight, an attribute object is created at the position where the light hits a nearby wall. The principal object of this attribute is of course the player. Now even if the player is hidden, but an enemy sees the attribute object (the light on the wall) he will in fact see the player! The rationale behind that is that the enemies have enough intelligence to interpolate where the origin of the light ray is coming from, and thus they immediately know the player's position. This is illustrated in the following diagram.

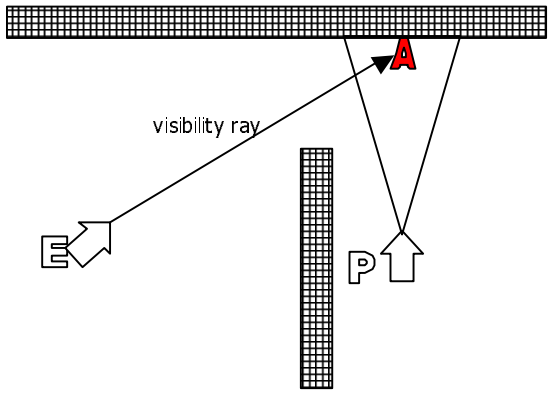


Fig. The enemy can see the attribute, even though it cannot see the player. Thus, he knows the position of the player.

The same feature is used when looking at rocks, grenades, rockets etc… Optionally it can be used to add more features to the game, for example if the enemies would leave footsteps on the ground that evaporate after some time, attribute objects can be spawned on the footsteps so that any guard who sees them will know where the person who left them is. The same can be expanded to blood tracks etc.

To make sure that the attribute objects are included in the visibility determination – they have to have an assessment multiplier set. Refer to "aiconfig.lua" in the scripts folder to see where Far Cry defines the multiplier for the attribute objects.

Perception coefficient and visibility

Visibility in Far Cry is not an on/off switch. Since the idea was to enable the player a certain freedom in choosing his playing style (action or stealth) it was necessary to come up with some mechanism that would allow the player to make certain amount of mistakes and still be able to recover from them.

To this end, every enemy has been given a perception value for every individual target it receives. This perception value describes how close an individual enemy is to actually seeing that particular target. When the perception coefficient

reaches its maximum value (currently 10) then the target has been seen. The initial value of the perception coefficient is 0 and the rules under which it can increase are discussed further.

The perception coefficient is only implemented in player AI objects. This is one of the reasons why a player AI object is specifically defined even in the lowest layer of the AI system hierarchy. What this means is that an enemy will not use the perception coefficient when perceiving another AI enemy – instead it will use "switch" vision – the target will be visible as soon as a ray can be shot to its position. There is of course a workaround and a way to declare that some AI enemy should also use a perception coefficient, but that will be discussed later.

When an AI enemy is processing a player target for visibility, if that player passes all possible visibility tests (including the visibility ray), it still has to go through one last test – the perception coefficient that the enemy stores for this particular player target has to reach maximum value before the enemy can receive a definite visual stimulus. This statement contains several corollaries:

1. Every enemy has his own perception coefficient for every player target it is processing.
2. Every enemy has to have his own perception coefficient reach maximum value before he receives notification that he sees a particular player target.
3. Perception coefficients of two different enemies (even for the same player target) are UNRELATED.
4. There is no GAME perception coefficient (a value that shows how much a player target is perceived by ANY enemy) – although this information can be derived by statistics.

The perception coefficient starts at 0. When the enemy starts receiving notification that some player target is passing the visibility determination routine, it starts adding to this value. The amount that is added to it depends on several factors – among others: the distance of the player target from the enemy, how high from the ground the player target is, if the player target is moving or static etc. All of these factors influence the rise of the perception coefficient in various ways.

Distance. This is the overwhelming factor in the rise of the perception coefficient. The closer the player target is to the enemy, the faster the coefficient rises – the farther away it is, the slower the coefficient rises. The increase function is a basic quadratic function as can be seen on the figure. At distances very close to the enemy, the time to reach the maximum perception coefficient is almost non-existent – the target is instantly seen. But it can be also seen that on the boundaries of the sight range sphere, the player can move freely as the rise of the coefficient is

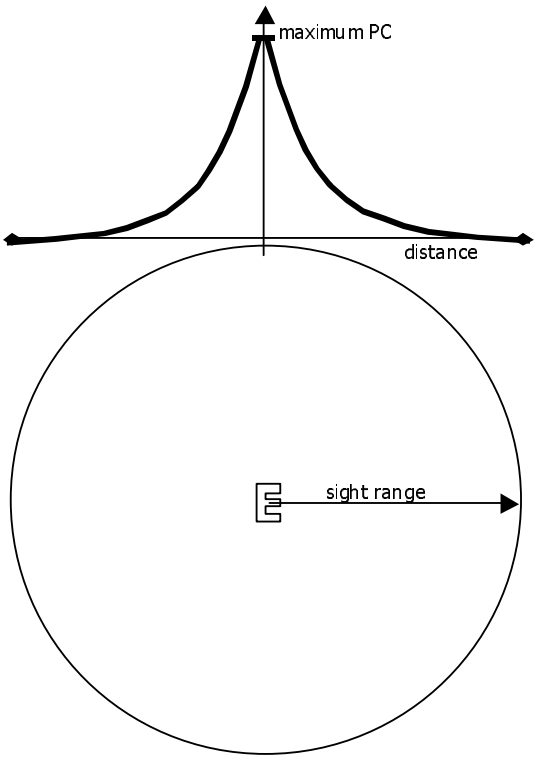


Fig. Speed of rise of perception coefficient as a function of the distance from the target.

very slow. As said before, the target STILL has to pass the complete visibility determination routine to affect the perception coefficient.

Height from ground. The player target's distance from the ground plane also is a major factor in determining the rate of increase of the perception coefficient. This is rationalized by the fact that a proning target is much harder to spot than someone who is standing up straight. The system has an idea of the distance of the target from the ground via the "eye height" property of each AI object. This property is set on initialization of the AI object and can be changed at anytime during the execution of the game. **Note**: If the enemies and the player have animated characters as their representation in the game, then the eye height is automatically calculated as the actual height of the head bone in the character. The influence of this property to the speed of increase of the perception coefficient is that its normal distance increase value (which is calculated as previously discussed based on the target's proximity to the enemy) is lowered by 50% if the target has a height less than 1 meter.

Target motion. It also makes a difference whether the player is standing still or he is moving. Obviously, stationary targets are much harder to spot then moving targets, so if the player is standing still, the rate of increase of the perception coefficient is lowered by additional 50% from the calculated value thus far.

Artificial modifiers. There are additional values that can define how fast the increase rate is. Some of them affect ALL enemies in the game world, and some affect only particular targets. An example of a modifier that affects all enemies is the console variable ai_SOM_SPEED. Its default value varies depending on the difficulty level, but it provides a constant multiplier that is applied on top of all other calculation to the rate of increase of the perception coefficient – for all enemies. On the other hand, it is possible to set a custom multiplier per object type that will be used as an additional multiplier when a certain target is processed for perception – this is however limited to the lowest level of the AI system and thus is not available for tweaking in Far Cry.

Obviously the effect of all these modifications on the coefficient is cumulative, so a target that is crouching and standing still will have an increase rate of only 25% of the calculated distance increase. The result is a floating-point value that is added to the perception coefficient during EACH FULL UPDATE. Basically, every time an enemy is fully updated, the update of any coefficients to any potential targets is done together with the visibility determination

Potential targets (now meaning targets with a perception coefficient bigger than 0) slowly expire as time passes. What that means is that if the perception coefficient is not constantly increased every full update, it will slowly fall to the default value of 0 over time. This can happen, for example, if a target was visible for a few seconds, raised the coefficient to 5 and then broke off visual contact again. The coefficient will slowly drop towards 0. This is implemented in order to reward players that tactically advance and then pause before continuing. They can ideally wait until the coefficient drops to 0 and then continue sneaking.

Finally, a statistical overview of the perception coefficients of all enemies for the player in Far Cry is given in the form of a HUD stealth-o-meter. This is the small gauge to the left and right of the radar circle in the HUD. It represents the highest perception coefficient of the player from all enemies that currently perceive him. What that means is that it basically ALWAYS shows the perception coefficient of the one enemy that is most likely to see the player. A full stealth-o-meter does not mean that ALL enemies see the player; it just means that there is at least ONE enemy that

can. An empty stealth-o-meter however means that at this time NO enemy can see the player.

Part IV – Tactical system

Anchors

An anchor is a **helper that designates a position** in space that is somehow special to the AI. It is used to help the AI when performing jobs (like fixing, smoking etc) and also to help the AI identify alarm spots, reinforcement spots etc.

You can **place an anchor using the AI button** in the editor. Click it and then select AI Anchor from the list. Now you can move over the map and click where you want to place this anchor.

Notice that the anchor has a distinct orientation – for some anchors, orientation is also important.

After you have placed it on the map, look in the properties for this anchor and you will find a field called Action. **Use the drop down dialog to select which action** to associate to this anchor.

You can also assign **anchor properties to a proximity trigger**. Look for the Action property in the properties of the proximity trigger.

Anchor actions

The anchor action can be chosen among one of the following:

Job anchors	
AIANCHOR_CLIPBOARD	AI inspects some apparatus, makes notes on clibpoard
AIANCHOR_FENCE	AI fix fence can also be used for other generic fixing
AIANCHOR_FENCE_LONG	AI fix fence for a long duration
AIANCHOR_HOOD	Car hood or anywhere else want to briefly stop and fix something on car
AIANCHOR_MUTATED	For Valerie in Volcano
AIANCHOR_PUSHBUTTON	Button on the wall for AI to push - head height
AIANCHOR_RAMPAGE	
AIANCHOR_SIT_WRITE	AI is sitting down writing stuff
AIANCHOR_SIT_TYPE	AI is sitting down typing
AIANCHOR_TOOLBOX	Location for toolbox eg. for fix car goes off to

	toolbox return to where he was working
AIANCHOR_WARMHANDS	AI warms his hands
AIANCHOR_WHEEL	Tighten wheel nuts etc for car
DO_SOMETHING_SPECIAL	
EXERCISE_HERE	
GUN_RACK	This is a place where there are guns to be picked up from
PLAY_CARDS_HERE	
PLANT_BOMB_HERE	
PUSH_THIS_WAY	
SHOOTING_TARGET	AI will chose this anchor as a target when practicing fire
SLEEP	
SWIM_HERE	
Jobs which use entity which can be bound to AI	
AIANCHOR_BEAKER	AI pours fluid between beakers, holds up looks at it
AIANCHOR_CHAIR	Swivel chair for AI to sit down, chair binds to AI so will move forward
AIANCHOR_DINNER1	Mutant chomping down on some corpse flesh
AIANCHOR_DINNER2	Mutant chomping down on some corpse flesh
AIANCHOR_EXAMINATION	AI conducts autopsy on a specimen.
AIANCHOR_MAGAZINE	AI sitting down picks up magazine and reads it
AIANCHOR_MICROSCOPE	Microscope for scientist AI to peer into - atm need props
AIANCHOR_PICKUP	AI can pick up crate etc and move to AIANCHOR_PUTDOWN
AIANCHOR_PUTDOWN	Place where the picked object must be put down
FISH_HERE	AI casts rod and fishes for a while
MUTANT_LOCK	
USE_RADIO_ANIM	
Idle anchors	
AIANCHOR_RELIEF	
AIANCHOR_SMOKE	
AIANCHOR_STAND_TYPE	Something interesting to look at on wall

AIANCHOR_SEAT	Place for AI to sit down. Seat does not bind to AI
AIANCHOR_OBSERVE	
AIANCHOR_FLASHLIGHT	
AIANCHOR_NOTIFY_GROUP_DELAY	
AIANCHOR_RANDOM_TALK	
AIANCHOR_MISSION_TALK	
MISSION_TALK_INPLACE	
MORPH_HERE	
INVESTIGATE_HERE	
SPECIAL_STAND_TYPE	
SPECIAL_ENTERCODE	
SPECIAL_ENABLE_TRIGGER	
SPECIAL_HOLD_SPOT	
SEAT_PRECISE	Place for AI to sit down. Seat does not bind to AI
Combat anchors	
AIANCHOR_PROTECT_THIS_POINT	
AIANCHOR_PUSH_ALARM	Scientist push alarm to bring guard
AIANCHOR_REINFORCEPOINT	
AIANCHOR_SHOOTSPOTCROUCH	
AIANCHOR_SHOOTSPOTSTAND	
AIANCHOR_THROW_FLARE	
BLIND_ALARM	
HOLD_THIS_POSITION	
HOLD_YOUR_FIRE	
MUTANT_AIRDUCT	
MUTANT_JUMP_SMALL	
MUTANT_JUMP_TARGET	
MUTANT_JUMP_TARGET_WALKING	
RESPOND_TO_REINFORCEMENT	
RETREAT_HERE	
RETREAT_WHEN_HALVED	If placed, it makes AIs retreat to the next RETREAT_HERE anchor when their number is halved

SNIPER_POTSHOT	
USE_THIS_MOUNTED_WEAPON	
Vehicles - boat spots	
AIANCHOR_BOATATTACK_SPOT	Attack spot for boat should be placed along beaches. When target is on ground, boat will use this spots to attack.

The following anchor types are automatically created by the game and must not be chosen for manually placed anchors:

Vehicle seat places	
AIANCHOR_BOATENTER_SPOT	Place to enter a boat seat.
z_CARENTER_DRIVER	
z_CARENTER_GUNNER	
z_CARENTER_PASSENGER1	
z_CARENTER_PASSENGER2	
z_CARENTER_PASSENGER3	
z_CARENTER_PASSENGER4	
z_CARENTER_PASSENGER5	
z_CARENTER_PASSENGER6	
z_CARENTER_PASSENGER7	
z_CARENTER_PASSENGER8	
z_CARENTER_PASSENGER9	
z_CARENTER_PASSENGER10	
z_HELYENTER	
Object related anchors	
AIOBJECT_DAMAGEGRENADE	
AIOBJECT_SWIVIL_CHAIR	
AIOBJECT_FLYING_FOX	
AIOBJECT_CARRY_CRATE	

PLACEHOLDER	Set as the object type for triggers

Part V – Scripting

Scripting in Far Cry doesn't necessarily mean creating predefined encounters that always happen in the same way – although you can do that just as easily. The concept of scripting here is meant more as a way to define how a particular AI object responds to a change in the environment – in essence: how it *behaves*. Thus, the scripts have the task to define the logic of the enemy's behaviour when the enemy receives any kind of stimuli – visual, auditory or in any other way.

The scripts are written using the script language LUA (http://www.lua.org/) and they take advantage of the unique properties that this scripting language offers. This section will assume that you have a thorough familiarity with writing lua script and will contain no introductory text for lua beginners. Refer to the link provided for exhaustive information on how to write effective lua script.

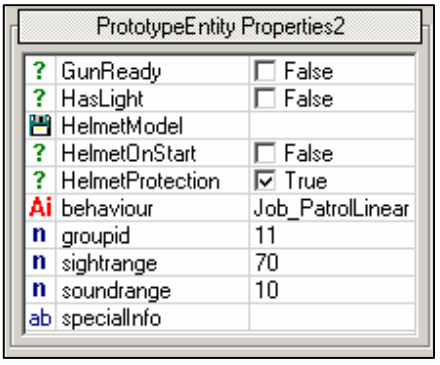
One important notice is that all the functionality exposed to lua in the form of script objects (tables) or functions is actually intrinsic C++ functionality. Anything that can be accomplished in lua can just as easily be ported to C++ code as all lua function calls have their C++ code counterparts. The choice of whether it is done in script or in code will have to involve all the pros and cons of each selection – the scripts will allow for quick turnaround and testing (since they can be reloaded runtime without restarting the game) but will undoubtedly perform slower than actual C++ code.

Behavior Scripts

All enemies have to have at least ONE behavior script before they can do anything in the game. The behavior script is the final authority that defines what a particular enemy has to do when something in his environment changes – like he sees a player, or he hears a sound or he receives a signal from a group member. All these environment changes translate to callback functions in the behavior script, and then it's the script's responsibility to define what the enemy's response will be.

When you place an enemy in the CryEngine Sandbox, it will come with some default behavior. But, if the enemy had NO behavior whatsoever, he will at minimum follow any targets he selects with his eyes, but he will just stand there. This means that target acquisition and processing is still working for this enemy, he just doesn't know what to do when he perceives a particular target.

Setting up the behavior



In the CryEngine Sandbox, the behavior for a particular enemy can be configured using the properties rollout that appears when the enemy is selected. This is shown in the picture. The behavior is selected by clicking on the "`Ai behaviour`" property and selecting it from the behavior selection dialog. At this time, we are not interested in any of the predefined

behaviors that can be selected in this dialog. In order to facilitate better introduction to the concept of the behavior, it's important to start with an empty behavior (an empty string will suffice in the place of the "Job_PatrolLinear" in the picture).

An enemy configured like this will never move or do anything of significance. The lowermost layer of the AI Object (sitting deep within C++ code) will still work, and that means that the enemy will still get visual and audible targets, but will just look at them and not do anything. If you try to move around this enemy in the editor's game mode, he will follow the player character by looking at him. Experiment with going closer and further from the enemy, as well as going to third person (F1) and crouching, proning or jumping in close proximity to the enemy. Observe how he will follow with his eyes, head and upper torso the position of the head of the player character.

The reason why this enemy is not doing anything is because when the AI system notified him that he is seeing a player target, there was no logic selected to determine what he should do in response to that. That is EXACTLY what the behavior should do – provide the system with instructions on what to do when something in the environment changes.

To select which behavior a particular enemy uses, the behavior name should be typed into the appropriate property ("Ai behaviour").

Creating a simple behavior

The behavior needs to be defined as a LUA table. There is a global LUA table called "AIBehaviour". This table contains (as its sub tables) all the behaviors that the game can use and that can be specified in the behavior property for an enemy. To be able to use a behavior in the game, it has to be registered as a sub table to this table.

The file "AIBehaviour.lua" in the "Scripts/AI/Behaviors" directory is a list of all possible behaviors in the game Far Cry. Ignoring the finer details of this file (like the existence of two sub tables "AVAILABLE" and "INTERNAL") we can add a new behavior to this table. For illustration, we can add the following line in the "AVAILABLE" sub table:

```
Prototype = "Scripts/AI/Behaviors/Personalities/Prototype.lua",
```

This line can be entered anywhere in the "AVAILABLE" sub table, but for the sake of argument, lets say we added it as the very first line after the "AVAILABLE = {"line. The script for this behavior EXISTS already (in the directory that you specified with the line you just added – look for it), but it is not selecting any response to any incoming event. It does however contain all the important events that can happen during execution of the game (even some events that are not really used ☺)– so the next step is to write some logic inside them in order to make the enemy that uses this behavior do something. Don't forget to save the "AIBehaviour.lua" after this and reload the scripts.

But first, let's look at the way events are described in the behavior in more detail. Looking at a part of the events in the Prototype.lua file, we might see something like this:

```
-----------------------------------------
OnNoTarget = function( self, entity )
      -- called when the enemy stops having an attention target
end,
-----------------------------------------
OnPlayerSeen = function( self, entity, fDistance )
      -- called when the enemy sees a living player
end,
-----------------------------------------
OnGrenadeSeen = function( self, entity, fDistance )
      -- called when the enemy sees a grenade
end,
```

The first thing to notice here is the fact that all of these "events" are in fact nothing more than functions with arguments. The name of the function is the name of the event, and the arguments (for most of the events) are always the same:

1. **self** – This argument represents the instance of the behavior itself. This sounds more complicated than it actually is. It basically provides a way inside the function to refer to something that is contained within an instance of the behavior. For example, if I would write

   ```
   self:OnPlayerSeen(entity,fDistance)
   ```

   inside the `OnPlayerSeen` function, then I would call the function I am in from inside it, making in essence a recursive call. Notice that when I use the colon (**:**) form of calling a function, the first argument "`self`" is implicitly passed – just like the "`this`" implicit argument in C++ classes. If I were to use the (**.**) method of calling the function, then I would have had to specify the "`self`" argument *explicitly*, like this:

   ```
   self.OnPlayerSeen(self,entity,fDistance)
   ```

   This is a strictly CryEngine behavior of LUA and its only mentioned here to avoid confusion about using the colon to call a function.
2. **entity** – Since behavior tables are loaded once and not reused, that means that ALL AI enemies reuse the same table. In order to be able to determine which entity is currently being processed, it's being passed as an argument to every event function handler. If you do not know or are not sure what an entity is and what its script table means and how it is used – please consult the CryEngine additional documentation.
3. Different functions can have different arguments after the initial two. While most of them don't we can even see in the `OnPlayerSeen` function that there is a third argument – the distance to the player target. The appendices contain a complete list of all system defined events and all arguments that are passed to them.

The behavior that was registered can be then selected from the behavior selection dialog, or just typed into the appropriate property of the entity. For the sake of clarity, we can manually type the behavior "Prototype" in the "behaviour" property – but to make sure that the behavior table script is reloaded, don't forget to reload the scripts in the editor (consult editor documentation on how to do that).

Responding to an event in the behavior

Now that the prototype behavior is set up, we can start experimenting with responding to some events that occur in the course of playing the game. In order to respond to any particular event, all we have to do is write some logic in the appropriate function of the behavior – for example if we wanted to do something when the enemy sees the player, we would write something in the "`OnPlayerSeen`" function of the behavior.

As an exercise, we will output some text when the enemy sees the player. This text will be outputted in the HUD, so you can confirm that the HUD is visible in the game mode of the editor by typing "`cl_display_hud 1`" in the console.

Next, lets replace the `OnPlayerSeen` function with the following version (the new parts are highlighted):

```
OnPlayerSeen = function( self, entity, fDistance )
      -- called when the enemy sees a living player
      Hud:AddMessage("I SEE YOU");
end,
```

Now, every time the enemy sees you, it will output this text to the console. To test this, save the new version of `Prototype.lua` and reload scripts in the editor. Then go into game mode and walk around the enemy until you see the text displayed in your HUD.

There are a few things to notice here – first of all notice that the event function is only called ONE time when the enemy actually perceives the player for the first time – NOT every frame afterwards. This is done to minimize the potentially very expensive script call when it's not necessary and when the script actually had a chance to act upon a change in the environment. The second thing to notice is that the enemy does NOT see you right away, but only after the stealth-o-meter rises to maximum value. This is normal operation and to understand how that works read this section.

The events are called ALWAYS when the appropriate event happens, even if multiple events happen at very close time intervals. To illustrate this, let's try to handle another system event that happens during the playing of the game – the `OnThreateningSoundHeard` event:

```
OnThreateningSoundHeard = function( self, entity )
      -- called when the enemy hears a scary sound
      Hud:AddMessage("I HEAR SOMETHING!!");
```



```
end,
```

To test this, enter game mode in a position behind the enemy that uses this behavior, and then shoot in the air. This will create the situation in which the enemy will hear the gunshot, thus receive the event that we just enhanced. After hearing the shot, he will turn to face the direction of the shot and consequently see the player. Then he will receive the usual "`OnPlayerSeen`" event.

There is no limit on what you can put as processing logic for a particular event. In the example we used a simple HUD message, but any kind of logic can be performed here. Having access to all global objects in the LUA state means you can affect the rendering, physical state of the world, play sounds or music or a particular animation, or send a message to another entity. This makes scripting AI response quite simple and quite powerful.

However, here we are not interested in affecting the world or indeed anything else then the entity that is executing the current behavior, since that is what is most interesting. To affect the entity and tell him what to do when a certain event is triggered, we need to get familiar with another concept in the AI system – goal pipes.

Goals and goal pipes

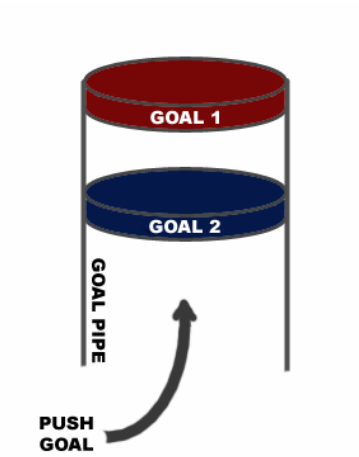
AI Objects in the CryAISystem are capable of executing certain actions in order to achieve some goals. The goals can be very simple – for example approach 2 meters to a certain object, or fairly complicated – for example hide in the leftmost obstacle from your current target.

The low level AI system is responsible for executing and management of these goals, and it also controls what happens if the goals successfully finish or if they fail. Other responsibilities include brokering which goals execute in parallel and which goals execute serially.

Goals can be atomic and derived. An **atomic** goal is a simple goal that cannot be further split into sub-goals, and is one logical unit. A **derived** goal is a goal that is a combination of a few atomic or derived goals that can execute in parallel or in a sequence. While atomic goals are defined by the system and introduction of new atomic goals would require working on the low level AI system, the user can *define* derived goals himself. The definition of new derived goals – from now I will call derived goals simply goals – can be made in the C++ side or in LUA (both contain a set of functions that work with goals). Usually, goals are exclusively defined in LUA script.

Since derived goals are created by combining atomic and other derived goals, there must be a way to organize a few goals into a logical union – a program – that has a start point, body to execute and conditions to end. This logical union exists in the CryAISystem and is called a **GOAL PIPE**.

Goal pipes



The logical union of atomic and derived goals for the purpose of creating another derived goal is called a goal pipe. It is called that because conceptually it resembles a pipe in which goals are pushed from one end, and as they execute they come out on the other end, and are re-pushed from the bottom again – like on the picture. This implies that the goals are executed in sequence – which is generally true – but there is a way to make goals execute in parallel.

Even a goal pipe can be pushed as a goal to another pipe. This enables such exotic concepts as nesting goal pipe, and recursive goal pipes. There is no limit of the amount of nesting with the goal pipes so that is left to the discretion of the user. Personally I would not recommend using recursion in pipes but the system generally supports it.

By combining atomic goals into bigger goals, you can develop any level of abstraction for the characters in the game. For example, exiting a room might involve finding a door and using it. So you have fragmented the higher level goal into two smaller goals. Finding a door decomposes into locating a door object and approaching to it – both of which are pretty atomic. Using the door involves approaching at 1 meter, and pressing USE. So while you can create one big goal pipe that locates a door object, approaches it, stands at 1 meter to it, presses use; you can also fragment the action into two higher level sub-goals and optionally reuse them sometime later. In the same sense, maybe you can use the "exit_room" goal pipe inside a bigger "exit_building" goal pipe etc. It is exactly for this purpose why the nesting of goal pipes is handy.

All goal pipes are created on startup. Goal pipes are not owned by any AI object, they are owned by the AI system itself, so you do not need to worry that you need to release them or otherwise maintain them. Every AI object that requests a goal pipe gets a clone of the original – that means that he has his own copy of the goal pipe and cannot possibly influence the operation of other AI objects that may use the same goal pipe. AI objects are the owners of these cloned goal pipes and they automatically perform maintenance on them, so it is another thing that doesn't need to concern the user.

To create a goal pipe, you have to call the following function (in LUA – there is an equivalent function in C++, but we will focus on LUA functions):

```
AI:CreateGoalPipe("crouchfire");
```

The AI global LUA script object offers this functionality (as well as other, which will be discussed later). The function is simple with its only argument being the name under which this goal pipe will be identified in the future.

Special care should be taken about naming the goal pipe. For one, it should not be a name of an existing atomic goal – a comprehensive list is included in the appendices. Ideally, it should be a unique name that can be easily remembered and it should pretty much explain what the goal pipe will do. Looking at the name of the

example goal pipe, it is easy to surmise that this goal pipe will make the entity that is using it to fire while crouching.

If an attempt is made to create a goal pipe with a name that already exists, the old version of the goal pipe will be orderly removed and cleaned up, and will be totally replaced with the new version. This doesn't mean that if someone was already using the old version he would not know what to do, or even use the new version. Goal pipes are CLONED to be used, so whoever is using an older version of the goal pipe will keep the cloned version of the old goal pipe and finish executing it, while whoever wants to get a new version of the goal pipe will have only the new version to clone FROM. In this sense, name conflicts for goal pipes are not necessarily a terrible thing.

Goal pipes do not need to be created ONLY on startup. The user is free to create a goal pipe at any point in the game's execution. The advantage to this is that you can create a goal pipe that depends on certain condition, or a goal pipe that is geared for a very specific use – one that may not be readily available at the startup of the game. Creation of goal pipes DOES mean memory allocation and access, but the overhead is not significant if used in moderation. A good rule of thumb is that goal pipes can be created in response to an event in a behavior because those events are generally not frequent. Creating new goalpipes every update frame is a bad idea.

Pushing goals in goal pipes

Once a goal pipe has been successfully created, you can start pushing goals into it. As mentioned previously, the goal pipe works in essence as an FIFO (first in first out) buffer. What that means is that the goals pushed into the pipe will execute in the same order in which they were initially pushed. There is no limit (other than common sense) to the amount of goals that you can push into a goal pipe and remember that you can also push sub-goal pipes into the pipe.

To push a goal into a pipe, you can use the following function:

```
AI:PushGoal("crouchfire","firecmd",0,1);
```

Notice that the function is a part of the functionality that the AI global LUA object offers. The arguments to this function are fairly self explanatory: first and foremost, the goal pipe in which we want to push has to be defined by name. Next, the actual goal that is going to be pushed onto this goal pipe has to be specified, also by name. In the example, that is the simple goal "`firecmd`" which basically instructs the enemy that executes it whether it is allowed to fire or not, based on its later parameters.

The third parameter of this function is an integer that can only be one of two values – 1 and 0. This is the last parameter that is common to all pushed goals – all parameters after this one are goal specific, which means that they are different based on the actual goal that needs to be pushed. This parameter is called the *blocking* flag. If it is set to 1, that means that when this goal starts executing, the further execution of the goal pipe should HALT until the currently executing goal finishes. If it is set to 0, that means that even though the current goal may not be finished, the execution of the goal pipe should continue.

This provides a mechanism to determine whether the goals will be executed in sequence or in parallel. Different derived goals require different behavior, so this is why this property can be controlled on a per goal level. For example, if we are writing a goal that needs to use an elevator button, it is normal that we need to wait before pressing use until we have approached the elevator to a certain distance. In this example, we would make the approach goal blocking, and follow it with a goal that generates a key press corresponding to the USE key.

Sub goal pipes CANNOT be non-blocking. They are blocking by default and any goal pipes that have sub pipes have to wait until the sub pipe has finished executing before they can continue with their own execution.

The appendices contain a full reference of all of the atomic goals that the system recognizes in its current state. Refer there for specific discussion on the parameters for a specific goal and its behavior.

The GoalPipes directory in the AI script directory contains all the goal pipes that were used in the game Far Cry, distributed over a few script files. Not all of those pipes were used. Refer to these files for an idea how finished goal pipes might look, and also for examples on the usage of the atomic goals, nesting etc.

Selecting Goal Pipes to execute

Once a goal pipe is created and goals are pushed into it, then it is ready to be selected by any entity that has an AI object for execution. Usually goal pipes are selected for execution in response to a game event and they specify to the entity that receives the event what it should do.

The selection of the goal pipe is the simplest and the most used function in scripting behaviors for the enemies. By selecting a particular goal pipe for a particular entity, you are forcing the entity to execute the goals contained within that goal pipe. One entity can execute ONLY ONE goal pipe at a time. When a new goal pipe is selected (and this is an asynchronous operation), the old pipe is removed and deleted, any remaining goals from the old pipe are also removed, and the new pipe is cloned and set for execution.

To select a goal pipe to execute for a particular entity, the following function should be called on the entity script object:

```
entity:SelectPipe(0,"cover_look_closer");
```

The entity script object is passed to every event handling function, so it is easy to call functions on it and change its state. The parameters of this call are very simple: the first integer parameter is reserved and should always be 0 (for partly historical reasons). The second parameter denotes the name of the goal pipe that needs to be selected. That has to be a name of an existing goal pipe; otherwise the function call will fail.

It makes no sense to make multiple calls to the SelectPipe function in a single event. Because the limitation that only ONE goal pipe can be executed at a time, only the last call to SelectPipe will actually be valid, and the last selected pipe will be the one that ends up executing.

There is a mechanism to asynchronously nest goal pipes into whatever goal pipe is currently executing for an entity. At any point in time, it is possible to INSERT a goal pipe into the currently executing one. This operation will stop the currently executing pipe; insert the selected goal pipe as its nested sub-pipe and continue execution inside the nested goal pipe.

To insert a sub-pipe into the currently executing goal pipe, the following function should be called on the entity script object:

```
entity:InsertSubpipe(0,"setup_stealth");
```

The sub pipe will always be inserted in the currently executing pipe, regardless of what recursion depth it may be currently at. What this means is that if the goal pipe was already executing a sub pipe (or more) at the time of the insertion, the new goal pipe will be inserted into the sub pipe.

Pipe insertion can be used in a wider context, for example to queue a few pipes to execute in some order. Calling InsertSubpipe multiple times in the same event does not have the same consequence as calling SelectPipe multiple times in the same event. The goal pipes that are inserted are always inserted at the top of the previous goal pipe, so when execution starts, the goals will be executed in a sequence starting from the goal in the last call to InsertSubpipe to the goal in the first call to InsertSubpipe. For example

```
entity:InsertSubpipe(0,"setup_stealth");
entity:InsertSubpipe(0,"DRAW_GUN");
```

will execute the goal pipe "`DRAW_GUN`" first and when its finished, it will execute the "`setup_stealth`" goal pipe. Once an inserted sub pipe is finished executing, it will return to the previous goal pipe in which it was initially inserted.

Goal pipe arguments and execution details

As discussed before, the goal pipes execute the goals pushed in them in sequence. But what happens when the final goal of the pipe is executed? The answer to this question depends on whether the goal pipe was selected or inserted.

Selected goal pipes will resume execution from the beginning if they execute to the end, meaning they wrap around. If a selected goal pipe is not in any way interrupted (by the selection of another goal pipe or by the insertion of a goal pipe) it will loop forever, together with any nested goal pipes that it has.

If however the pipe was inserted, after it executes the last goal, it will be removed, deleted and will not be executed again unless it is inserted again. This is valid ONLY for pipes that are inserted asynchronously – if a goal pipe is NESTED in some other goal pipe by pushing it in the normal procedure, it will be re-executed every time the parent pipe reaches it.

Goal pipe execution is pretty robust. It is natural that some goals require some prerequisites to run properly, and most of the time these prerequisites will be met – for example, the approach goal cannot execute without an attention target,

because it will not know to what it should approach to. If situations like this occur during the execution of the goals, the goal is suspended (finished) and execution in the goal pipe continues uninterrupted. The consequences of this are sometimes positive and sometimes negative, depending on the context of the pipe execution and the desired behavior. Knowing the default operating procedure of goals that cannot execute properly should help the behavior designer in diagnosing problems and coming up with solutions.

Goal pipes can also **receive an argument**. Only one type of argument is allowed for the goal pipes, and that is a valid AI object – it can be an entity, a tag point, and anchor etc. The argument is passed to the pipe at selection (or insertion) time, and it is placed in the last operation target operand. In there it can be easily accessed by any goal executing in the pipe since a lot of goals can be configured to operate on the last operation target instead of the attention target. A discussion on attention targets and last operation targets can be found here.

Goal pipes remember their argument, which means that it is possible to nest different goal pipes with different arguments. The system will make sure that always the correct argument is placed into the last operation target before an inserted or selected pipe starts to execute.

To specify an argument to a goal pipe, you have to supply it during the call to SelectPipe or InsertSubpipe. The argument can only be a valid AI Object, but the way it is specified can be done in multiple ways. For example, you can specify a target entity as an argument by passing its ENTITY ID to the selection function, or generally any AI object by specifying the NAME of the desired AI object. The AI system guarantees that names of AI objects will be unique – but not necessarily exactly what the user named them. In that sense, it is not recommended to "hardcode" names into the arguments – instead they should be retrieved and just passed around.

The argument passing mechanism is robust enough and it will handle any strange situations, like for example when a name of an argument is passed that it cannot find. The execution of the goal pipe will continue normally.

Here is an example on how to specify an argument to a goal pipes:

```
OnGroupMemberDiedNearest = function (self, entity, sender)
	-- someone from your group died
	entity:SelectPipe(0,"RecogCorpse",sender.id);
end,
```

In this example, the pipe "RecogCorpse" is selected and the sender of the "OnGroupMemberDiedNearest" signal is passed as an argument to it using its entity id. This signal is sent by a dying enemy, and is received by the team member closest to his position. The receiver of this signal probably will try to move towards the sender in order to check if he is still alive.

Another example, that illustrates passing the argument to the pipe via its name:

```
local gunrack=AI:FindObjectOfType(self:GetPos(),30,AIAnchor.GUN_RACK);
if (gunrack) then
	self:InsertSubpipe(0,"get_gun",gunrack);
```



```
end
```

The function `FindObjectOfType` will return the name of the closest object in the 30 meter radius of type `AIAnchor.GUN_RACK`. Notice that we never handle the name explicitly, it is just passed to the pipe insertion function "as-is" returned from the Find function. Also notice that now the entity is referred as `self`, instead of `entity`. This means it is probably called from within its own script, and not the script of the behavior.

Combining behavior scripts and goal pipes

Now that we have covered the basics of how the behavior works and how events are handled in the behavior as well as how to create goals that can be used to respond to these events, it is time to combine the two and make a simple behavior that illustrates all the concepts.

This part of the manual will demonstrate how to write a simple behavior script. For real world examples, please refer to the Far Cry behavior scripts – which are nothing more than more complicated instances of behaviors. For the following example, I will use goal pipes that have already been written for Far Cry – to find where they are defined, please search the folder "`Scripts/AI/GoalPipes`" in your Far Cry directory.

Setting up a situation

In order to make a fairly illustrative example, we need to set up a situation in which the enemy will be engaged. This is an easy task – in the CryEngine Sandbox place a few brush objects (rocks, crates, whatever) at some close proximity to each other. Make sure that their "`Hideable`" flag is set, and then generate the AI triangulation. Now the enemy that we are going to place here will have somewhere to hide, which means we can use goal pipes that have the "`hide`" goal pipe within them. The final result should look something like this:

Create an enemy and select the already registered `Prototype` behavior – if this behavior is not registered for you, please revert to this section to understand how to register it. After selecting this behavior, try to engage the enemy in the editor's game mode. Since this behavior is essentially empty, the enemy just follows you with his eyes as you move around.

Implementing reasonable responses

Now that the situation is set up correctly, we can start defining what the enemy's reactions will be when engaging the player. For this purpose we will start with the simplest response an enemy character can have towards a player – shoot.

Replace the `OnPlayerSeen` function handler in the Prototype.lua with this one:

```
OnPlayerSeen = function( self, entity, fDistance )
      entity:SelectPipe(0,"just_shoot");
end,
```

This handler instructs the enemy to select the goal pipe "`just_shoot`" when he perceives a player. It's not hard to guess what that goal pipe is actually doing, but lets look at its implementation (can be found in the file `PipeManager2.lua`):

```
AI:CreateGoalPipe("just_shoot");
```

```
AI:PushGoal("just_shoot","firecmd",0,1);
AI:PushGoal("just_shoot","timeout",1,1);
```

This is really a simple goal pipe that allows the enemy to shoot with the first goal, and waits for 1 second with the second goal. Arguably, the timeout is not really necessary, but its there to make this pipe a little more than ONE simple goal ☺.

When executing this pipe, the default nature of re-executing the goal when it reaches the final goal is not a problem. Generally, an effort should be made to make the goal pipes that are to be selected "loop-able". In other words, they should continue to loop if nothing happens and the resulting behavior should not look out of place.

Experiment with spawning behind the enemy, and understanding when he starts to actually use the "`just_shoot`" pipe. Notice that he may turn to you before you actually make visual contact with him – this just means he heard your footsteps.

Let's now expand this simple behavior by implementing a little more "logic" in the event handler. For example, we might want the enemy to run for cover if the player is too close to him but just shoot if the player is far enough. To do something like this, we can change the event handler to something like this:

```
OnPlayerSeen = function( self, entity, fDistance )
      if (fDistance>5) then
            entity:SelectPipe(0,"just_shoot");
      else
            entity:SelectPipe(0,"minimize_exposure");
      end
end,
```

You can find the "minimize_exposure" goal pipe content in `PMReusable.lua`.

If your enemy is not hiding, make sure the proper hidable flags are set to the brushes and that you have generated the AI triangulation. Read this.

There are a few points to notice with this change. First, if you spawn at a distance greater than 5 meters and the enemy will start just shooting at you. This is good and expected. But, if you start approaching the enemy, and even reach a position closer than 5 meters to him, he will not select the goal pipe that makes him hide. Why is this happening?

In the previous discussion of events ant the mechanism for their triggering it was mentioned that events are ONLY generated when there is a change in the environment. Since once the enemy has seen the target there is no further change (he still sees the target) even if the target moves closer to the enemy. To make the enemy re-evaluate his position through the script, the environment must change (e.g, the player hides and then comes out again – you break visual contact with the enemy). To understand this more clearly, let's reintroduce the HUD messages to see when the event handler functions are being called:

```
OnPlayerSeen = function( self, entity, fDistance )
      Hud:AddMessage("On Player Seen");
      if (fDistance>5) then
```

```
                entity:SelectPipe(0,"just_shoot");
        else
                entity:SelectPipe(0,"minimize_exposure");
        end
end,

OnEnemyMemory = function( self, entity, fDistance )
        Hud:AddMessage("On Enemy Memory");
end,
```

Now try to run around and see when the messages are displayed. Notice that every time the enemy turns his back to you, the `OnEnemyMemory` event is called – since the environment changed and the enemy cannot see you anymore. To reissue the `OnPlayerSeen`, notice that there must be an `OnEnemyMemory` in between that will cause a re-evaluation.

There is a way to force reevaluation of targets, but that's not recommended since reevaluation is the most expensive thing the AI object can be doing and therefore there are a lot of automatic mechanisms that ensure that it is done as seldom as possible. To force reevaluation at any point in a goal pipe, you can use the goal "clear".

Let's try to expand upon this simple behavior, in that we will make the enemy first run to hide, and then shoot from there – just like he did previously. The simplest way of accomplishing this is to insert a pipe at the beginning of "`just_shoot`" that will make the enemy go to cover. To do that, change the `OnPlayerSeen` event handler to look like this:

```
OnPlayerSeen = function( self, entity, fDistance )
        entity:SelectPipe(0,"just_shoot");
        entity:InsertSubpipe(0,"minimize_exposure");
end,
```

Notice that before inserting a pipe in to the currently executing pipe – a currently executing pipe has to be selected. Switching the order of the function calls will not work.

Now try testing this, and you will observe an interesting and unexpected turn of events. If you expected the enemy to hide one time and then just shoot from there, that is obviously not what is happening. The enemy keeps running back behind cover, even after the first time, and he doesn't shoot much. Why is this happening?

The answer is the same as the previous time. The fact that the enemy TURNS HIS BACK to the target when he turns to hide, makes him receive the `OnEnemyMemory` event, thus causing a reevaluation. Then when he looks back towards the target, he gets the `OnPlayerSeen` event again which causes the "`minimize_exposure`" goal pipe to be inserted again. Thus the enemy keeps hiding.

How to resolve this? In order to realize what is needed to fix this "problem", we need to understand that the response that the enemy does when he sees the player FOR THE FIRST TIME is not necessarily the same response he will do when he

sees the player again. This is because the enemy was previously idle, but after seeing the player, he switches to alert mode. Essentially, the enemy has changed the state, so he needs to start responding to events differently.

But how to change state when we have only one behavior? Well, think of behaviors as STATES, and multiple states can be created by creating multiple behaviors. In the presence of multiple behaviors, there must exist a mechanism for deciding WHEN certain behavior needs to go into another behavior (when states need to change), and the rules that define to what states the enemy can switch from any given state. Just such a mechanism is the enemy CHARACTER.

Character Scripts

When making a separate behavior script for every state that the enemy can be in (idle, alert, attack) there must be a way to determine the following important properties:

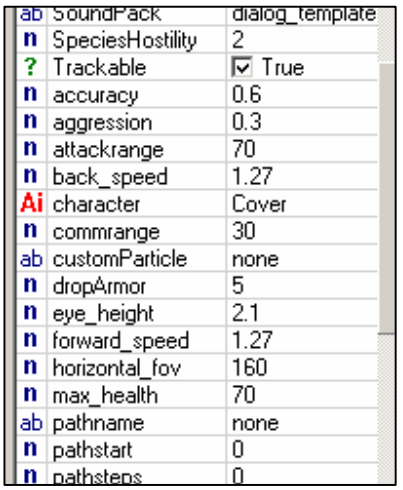
1. Which behavior is the CURRENT behavior? All incoming game events have to be handled in the current behavior by default. Thus, the current behavior has to be defined at all times.
2. What needs to happen in order for the current behavior to change? There must be a set of well defined rules that indicate how the behaviors change. For example, if the enemy was in the idle behavior, and he saw the player, then he needs to move in the attack behavior. In this case, the current behavior is the idle, the event that triggers the change is the seeing of the player, and the new behavior that needs to become the current one is the attack behavior.

This is exactly the purpose of the Character scripts. They define the actual rules for changing behaviors (states) and can be found in the `Scripts/AI/Characters` folder in your Far Cry installation directory.

The Character script concept

The character script is a completely separate script file from the behavior script. While it is possible to create enemies that have a single behavior script (this is for example the Pig in Far Cry), when using multiple behaviors a valid Character script must be selected.

Selecting the character for an enemy entity to use is similar to selecting the behavior. The only difference is that the character property is not a "per instance" property for the enemies, but a "per archetype" property. If you are not using archetypes, then it doesn't matter for you, but if you are then you have to access the archetype properties before you can change the character property.



The character property is called "AI character" (as can be seen in the picture). It also has an automatic selection dialog – just like the behavior. The entries listed in this dialog are all available characters that can be selected and that are properly registered within the system. Alternatively, the character name can by typed as a string, or the property can be an empty string in which case it means that the character is not necessary. This can be the case when a single behavior enemy is used.

The character scripts DO NOT contain any script logic. They are strictly data scripts and represent tables that are looked up periodically to determine whether the enemy should change behavior, and if so to determine to which behavior the enemy should change to.

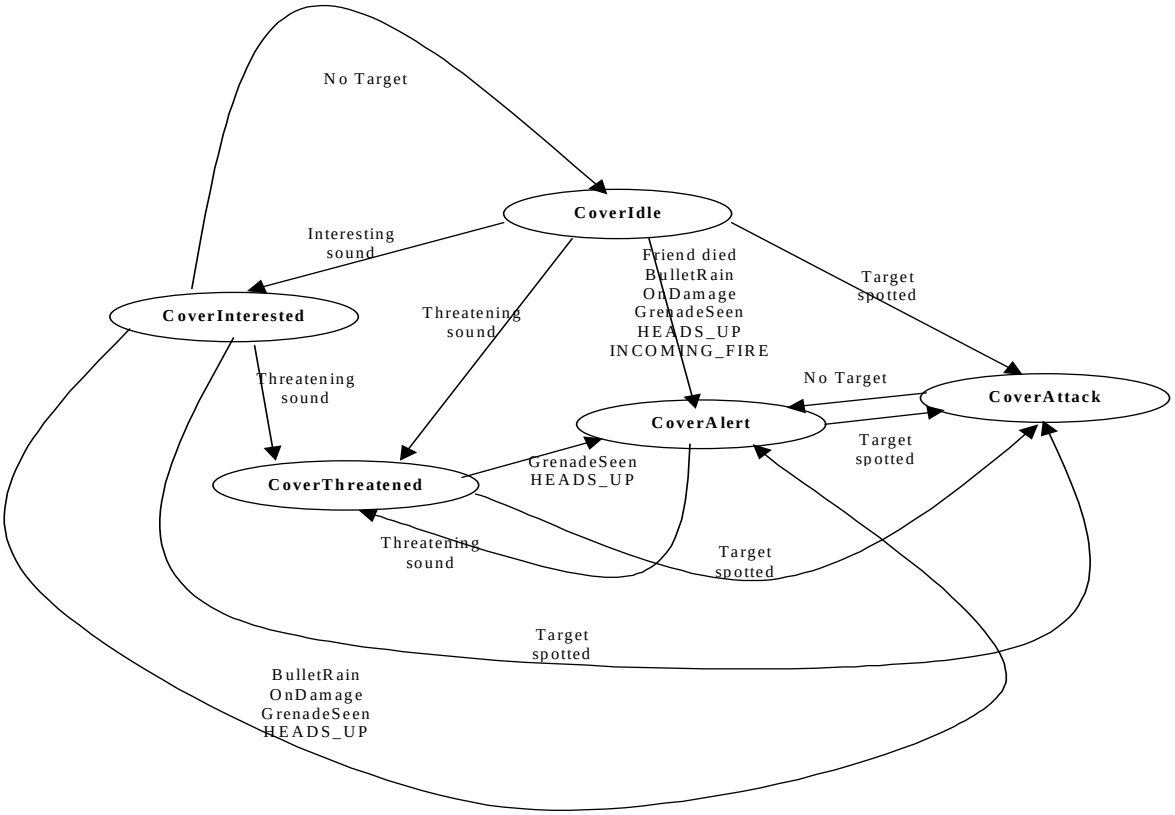
A character script cannot be used in the game unless it has been registered inside the `AICharacter.lua` script file. This file is very similar to the `AIBehavior.lua` file in that it specifies the character script table name and the full path to the actual script that contains that table. This file can be found in the `Scripts/AI/Characters` directory, and this is how it might look like (this is not the entire file:

```
AICharacter = {

	AVAILABLE = {
		Cover		= "Scripts/AI/Characters/Personalities/Cover.lua",
		Scout		= "Scripts/AI/Characters/Personalities/Scout.lua",
		Swat		= "Scripts/AI/Characters/Personalities/Swat.lua",
		Rear		= "Scripts/AI/Characters/Personalities/Rear.lua",
		Sniper		= "Scripts/AI/Characters/Personalities/Sniper.lua",
		....
```

The characters that are listed in the character selection dialog all come from this table and are enumerated every time the editor restarts or scripts are reloaded.

Conceptually, the character script can be represented graphically as a connected graph. The nodes of the graph are the behaviors and represent the states that the enemy can be in. The links of the graph are the events that need to happen in order for the enemy to move into another state (change behavior). A schematic like that has already been represented in the previous text, but here it is again for clarity:



In light of the new concepts of behaviour and events that we introduced in the previous text, this picture is starting to make more sense now. All the nodes will infact correspond 1:1 to an actual behaviour Script, and all the links correspond to actual in-game events. And indeed, if you check the `Scripts/AI/Behaviors/Personalities/Cover` directory, you will find all of the scripts mentioned in the diagram.

From the diagram we can see that if the enemy was in the `CoverIdle` behaviour, and received the signal "interesting sound", then he needs to change his current behaviour into the `CoverInterested` behaviour. All subsequent events that happen will be handled in the `CoverInterested` behaviour until such an event happens that triggers a change out of this behaviour and into another. Obviously the responses of the enemy in the "interested" behaviour would be slightly different than those in the "idle" behaviour.

A diagram like this was an essential part of planning the enemy behaviours in the game Far Cry. Before doing anything else, each enemy had to have a diagram like this and that provided a mental image of how this enemy would behave under different circumstances. In other words, getting this diagram was the hard part – translating it into a character script was easy.

The character scripts contain entries in a very simple form. They contain a table for each of the behaviours that the enemy using the character can be possibly in. Each of those tables then contains a list of all events that can trigger a change into another behaviour. Together with the actual events, the new behaviour is also specified. So in general terms, a character script contains entries of the following form:

```
<current behaviour> = {
      <signal> = <next behaviour>,
      <signal> = <next behaviour>,
      …
}
```

The current behaviour represents any behaviour that the enemy can be executing at some time – this corresponds to the nodes in the graph in the picture. The signal represents any incoming signals that can cause a change into another behaviour – this corresponds to the links in the picture. And finally – the next behaviour corresponds to the behaviour that the enemy needs to switch to after receiving a given signal – this corresponds to the node that the followed link points to.

Let's illustrate this with an example taken directly from the Cover graph. Lets look at how the entry for the `CoverInterested` behaviour would look like. We can see that if we receive the signal "No Target", we need to switch to `CoverIdle`. This can be represented like this:

```
CoverInterested = {
      OnNoTarget = "CoverIdle",
}
```

The "OnNoTarget" game event is the accurate name of the event that occurs when the enemy had a target previously, but now lost it. The diagram does not contain the actual names of the events, just their conceptual names.

Further, we see in the diagram that if we receive a signal that tells us that a target was

seen, the behavior needs to change into `CoverAttack`. Similarly, if a threatening sound was heard, then the behavior needs to change into `CoverThreatened`. Let's add this into the entry:

```
CoverInterested = {
      OnNoTarget                  = "CoverIdle",
      OnPlayerSeen                = "CoverAttack",
      OnThreateningSoundHeard     = "CoverThreatened",
}
```

The final behavior to add is the CoverAlert behavior. Here we see an interesting change, in that multiple events can cause the enemy to switch to this behavior. There is no limit on how many events can be used to change the behavior; it only depends on what the desired behavior logic is. Let's add all the remaining events to the table entry:

```
CoverInterested = {
      OnNoTarget                  = "CoverIdle",
```



```
        OnPlayerSeen                    = “CoverAttack”,
        OnThreateningSoundHeard         = “CoverThreatened”,
        OnBulletRain                    = “CoverAlert”,
        OnReceivingDamage               = “CoverAlert”,
        OnGrenadeSeen                   = “CoverAlert”,
        HEADS_UP_GUYS                   = “CoverAlert”,
    }
```

The last entry is very interesting, as it is not a signal that the system generates, but a custom signal created by the user. In this case it happens when one enemy from a group of more enemies sees any kind of disturbance, he sends the signal “HEADS_UP_GUYS” which causes the rest of his team to become alerted.

This means that ANY signal, system or user defined, can be used to trigger a change in behaviors. In this way, some really specific and powerful behaviors can be created.

There is no limit on the amount of behaviors that can be created for a particular enemy type and no limit on the amount of entries in the character table.

The mechanics of changing behavior

To be able to fully use the vast amount of customizability and power that the character scripts provide the user, a thorough understanding of how they work and where do they fit in the normal game loop is required. This small section will discuss exactly that.

Characters are loaded on startup, and each enemy that uses a particular character is given a reference to it. Character scripts, like behavior scripts are NOT copied for every enemy that wants to use them. There is one instance of a particular character that ALL enemies that use that character refer to.

Every time a game event arrives down to the logic layer (the script layer) in order to process the event handler, the system automatically checks if this particular event needs to trigger a change in the behavior. The actual order of events is as follows:

- An event arrives for processing
- The event function handler in the current behavior is called and the function is executed. This is important since regardless of whether this particular event will trigger a change in behavior or not – the contents of the handler function MUST be executed.
- The system checks whether a character script exists for this enemy
- The character script is analyzed if it contains an entry for the current behavior
- A search is performed inside the current behavior entry for the event that is processing
- If such an event is found, then the behavior is changed to the one specified with the event

Of course, how much of these steps will be actually executed depends on the result of some of the searches performed. There are some steps that are taken even after the final step indicated here, but they are discussed in detail later. The sequence of events as depicted here is the normal operation for an event that will actually trigger a change of behavior. It is perhaps useful to mention that a large majority of the events will not actually trigger a change.

It is also worth to mention that the same mechanism happens every time a user generated signal is processed. That enables behavior changing on user defined signals.

All the operations specified here are on a script level, and as such are not overly expensive. This means that generally changing the behavior around is not discouraged, but it may lead to problems in maintenance of an enemy character that has many rules of how if changes behaviors. For the sake of clarity and your own sanity, you should keep the behavior changing to the absolute minimum possible.

Making a new character

To make a new character, there are a few extra steps that need to be taken apart from actually typing the new script. And even in typing the script, there are some guidelines that need to be followed.

The new character script needs to create a NEW script table, and declare it as a sub table to the `AICharacter` script table. The `AICharacter` script table is always created and it is global. It should contain ALL character scripts that will be used in the game, and no character script will be loaded unless it is declared as a sub table to this table. The declaration is rather simple, and it should look something like this:

```
AICharacter.Cover = {
}
```

This declares the "`Cover`" script table under the `AICharacter` script table.

Doing just this will not make it possible to select the character for an enemy inside the game. In order to do that, the character has to be registered with the system and shown the location of the new script file so it can load it. This procedure is very similar to the registering of the behaviors. The file that keeps a list of all the registered characters and their corresponding script files is the `AICharacter.lua` file which can be found in the `Scripts/AI/Characters` folder.

The format of an entry in this file is very similar to the registration entry for behaviors. It basically lists the name of the script table together with the path to the script file in which this table is declared:

```
<table name> = <path to script file relative to game root folder>,
```

These entries are declared in the `AVAILABLE` sub table of the `AICharacter` table. For example, if you have created a new character called "`InfantrySoldier`" and saved the file containing it in the file "`Infantry.lua`" in the Characters folder, then this is how its registration entry should look like:

```
AVAILABLE = {
      …
      InfantrySoldier = “Scripts/AI/Characters/Infantry.lua”,
      …
},
```

A character registered in this way can then be specified by typing its table name into the character entity property for enemies, or it can be selected from the character selection dialog in the Sanbox.

Relationship between behaviors and characters

There is no STRONG relationship between characters and behaviors. In theory, you can freely mix and match any character with any behavior – so really there is no conceptual pairing between characters and behaviors.

In the loosest possible terms, a character only depends on the behaviors that it has specified inside it. In Far Cry, for example, there is a big amount of shared behaviors that are then reused inside almost all of the characters. If a user would make a character that specifies that an enemy should go from `CoverIdle` into `ScoutAttack` when it spots an enemy – this is by no means an error situation – conceptually.

On the other hand, behaviors are not at all aware of the character existence. As far as the execution of the behavior goes – it is irrelevant whether a character exists at the same time or not – it has no bearing on the behavior execution. It is however possible that the behavior implements certain logic inside an event that can cause different signals to be emitted and thus “influence” implicitly to which behavior the character will switch. This is also not an error condition – in fact, a few of the Far Cry game behaviors employ this trick.

One thing that is VERY important to understand in regards to characters, behaviors and behavior switching is the following: If the character specifies that the behavior needs to be switched on a particular event – THERE IS NOTHING that the behavior can possibly do (overall and inside the function handler) to prevent that. In other words, it is impossible to implement any logic that uses the same event to switch to one behavior if some condition is true and another behavior when the same condition is false. Then what is the solution for this?

The solution is to create special signals for the two behaviors, and use them as the events that will trigger the behavior switch instead of using one event. As a simple example to illustrate this, consider the following:

```
----------------------------------------
      OnThreateningSoundHeard = function( self, entity, fDistance )
            if (fDistance<40) then
                  AI:Signal(0,1,"HOLD_POSITION",entity.id);
            else
                  AI:Signal(0,1,"NORMAL_THREAT_SOUND",entity.id);
```

```
        end
    end,
```

So now, instead of performing the behavior change on the OnThreateningSoundHeard event, we implemented some kind of logic inside it that determines which signal will be sent. The change to the different behaviors is done on the special signals.

In this way it is possible to define with logic from the current behavior what the next behavior will be. The example is a condensed real-world example from the Far Cry behaviors.

Implementation and execution details

Having covered the basics of behaviors and characters (and then some), the only thing remaining is to go over some of the nasty implementation details and some of the additional power afforded to the use via some clever detail features.

This section is very important to read since the details presented here are essential in being able to squeeze the maximum out of the system, and devise the simplest solutions to any problems that the user may have.

Behavior call indirection levels

This rather confusing title has to do with a very cool and advanced feature that the behavior and character processing have. To understand the necessity of call indirection, I will present a simple example.

Let's say that you are creating a set of behaviors and a character for a simple infantry soldier. A pretty good implementation would contain a few behavior scripts (idle, attack, retreat for example) and a character script. Now, since you have multiple behavior scripts, you have to handle all events that can possibly happen during playing in all behavior scripts. Even if you wanted the response to "`OnReceivingDamage`" to be the same across all behaviors, you would still have to copy/paste an event handler in every one of the behaviors, and if you later wanted to change something, you would have to manually change it in all behaviors.

Someone might say – so what? A little work is not a problem. It turns out, however, that in reality multiplying work is not such a good thing, not only for the additional time it takes, but also because of the difficulty in maintaining such a redundant system. This is why call indirection was introduced.

What is call indirection? The core idea is that it is not necessary to implement ALL events in ALL behaviors, but that the behaviors only implement the events that it is interested in, and the rest would be handled by some sort of fallback default behavior. This would minimize code duplication and would make the script code easier to maintain.

The question is, what is this default behavior, and how many levels of indirection should there be? While in theory it is possible to have as many levels of

indirection as you want, in the implementation of the Far Cry behavior and character system we opted for 2 (two).

One final side note before we examine how the indirection works in detail – in order to have indirection in the event handling, you must have a character for your script logic – essentially a system of multiple behaviors. If you are using a single behavior for the whole logic of some enemy, you do not have indirection – and you probably don't need it.

When a game event is registered and needs to be processed, the system checks to see whether a handler for this event is present in the currently executing behavior. If so, the function handler is executed and the story ends there.

However, if there isn't a function handler for this event in the currently executing behavior, then the system tries to find a function handler for the same event in the so-called "default character" behavior. This behavior is the default behavior FOR THE SELECTED character. It is always the behavior that has the name of the character with the word "Idle" appended to it. For example, the default behavior of the `Cover` character is the `CoverIdle` behavior, the default behavior of the `Fast` character is the `FastIdle` behavior, etc.

If the event is handled in the default character behavior, then its function is executed and the story ends *there*. As a result, a cursory examination of the behaviors that are actually used in Far Cry will uncover that the majority of the handlers for the events are in the behaviors that end on the word Idle – meaning in the default character behaviors.

Finally, if an event handler cannot be found in the default character behavior, the system tries to find a handler for the event in the DEFAULT behavior. This is the default behavior for ALL possible characters, and denotes the behavior that everyone falls back on if their behaviors do not implement a certain event handler. This behavior is a single behavior that can be found in the `Script/AI/Behaviors` folder under the name `DEFAULT.lua`. This script file is pretty large since it contains all the processing common to all AI characters in the Far Cry game.

If a handler for a particular event is not found even in the default behavior, then the event is simply discarded – no one was interested in acting on it. This does not happen ONLY with system events (like `OnPlayerSeen` and such) but also with ALL other user defined signals.

To recap, here are the implications of behavior call indirection:

1. All behaviors do not need to handle all events – only the ones that they are interested in;
2. If some event needs to be handled the same across all behaviors for a given character, then it should be implemented in the default character behavior;
3. If some event needs to be handled the same across ALL BEHAVIORS, then it should be implemented in the default behavior (`Default.lua`).

This introduces another "problem". What if some behavior is not interested in a particular event, but at the same time does not want default processing for that event? Then it is not enough to just ignore the event – the behavior needs to not only ignore the event but also stop it from propagating into the default behaviors. As it turns out – the solution to this problem is trivial: If the behavior wants to stop the

event from default processing, all it has to do is "sink" it – in essence provide a function for handling the event, but it should be an empty function. Since the system cares only if the function to handle the events EXISTS or not, and not whether it does anything, if an empty function is specified, the event will be effectively ignored.

Far Cry scripts use these levels of indirection and sinking of events in some behaviors very frequently. It has made script maintenance and bug fixing a much more enjoyable experience. I recommend you do the same. Experiment with indirection and use it as you would use inheritance ☺

Character indirection levels

You knew it was coming. Just as the behaviors have indirection levels, so do the characters. The motivation for introducing them in the execution of the character scripts is the same as the motivation to put them in the behaviors – ease of maintenance, elimination of duplicate code etc.

The problem of duplication is not so acute in the character scripts since they usually do not contain as much text as the behaviors, but the ease at which new behavior changes pertaining to every AI object in the world can be introduced was alone worth the implementation time.

There is only one level of indirection for character scripts, but it is a little more detailed than the behavior indirection. In essence, when an event has been processed in some behavior, the system searches whether this behavior can be found in the character script specified for a particular enemy. If a behavior entry can be found in the character table, the processing ends there.

If the behavior is not found, then the system tries to find an entry for the same behavior in the DEFAULT character. The default character is very similar to the default behavior and it can be found in the `Scripts/AI/Characters` folder in the `DEFAULT.lua` file. If a behavior entry in the default character is found, then the processing ends there.

Finally, if a there is no entry for the specified behavior, the system opens the `NoBehaviorFound` entry, and looks for the event there. In essence, if there isn't a behavior entry in the normal or the default character, the system will always look for the currently processing event in the `NoBehaviorFound` entry. This enables the user to guarantee that some signals will ALWAYS switch to some specified behavior regardless of what the currently executing behavior is.

Character indirection is also heavily used in Far Cry scripts. Its quite a powerful concept and it makes the scripter life easier.

General logic scripts

Behaviors and characters are not the only types of scripts that exist in the AI portion of Far Cry scripting. Since LUA is a very flexible scripting language, and one that offers instant access to any global objects and global functions, it is easy to abstract a lot of the common functionality in a separate LUA table and write that code only once but call it from all places where it is needed.

In Far Cry, this strategy is used very often. While the actual implementation details can be changed at any time and by anyone, this part will provide a little

insight on how the general logic scripts were implemented for Far Cry, and it might give the user an idea of the scope of things that can be done in this way.

As a general guideline, anything that needs to be called by any behavior or script on more than one occasion – it is a good idea to isolate it into its own special table. Some of those general logic tables can be found in the Scripts/AI/Logic folder in you Far Cry installation directory.

The idle animation manager

In a lot of places, during the execution of an idle behavior or any kind of job, the system needs to select one idle animation (head scratching, arm stretching etc) from some pool of existing animations. There are a certain amount of fine details when it comes down to how these animations are selected, how the system knows from which animations it can choose etc. This piece of script logic addresses all these issues.

There are a couple of functions that the Idle manager implements, and they basically have to do with populating a list of available animations and selecting one of them at semi-random. That means that while the selection of the animation to be returned is random, the system makes sure that all animations are shown before the same animation is seen again. However, the order in which the animations are shown is indeed random.

The function that selects the controlled random animation is the `GetIdle` function. It receives the name of the model for the specified enemy and then analyzes whether there are animations for this model. If there are, it selects them at random, and tags the ones that have been selected already. Every random selection is made from animations that have not been tagged. When there is no untagged animation left, the tags are cleared on all of them, and the process starts again with randomly selecting an animation. Please refer to the actual script for the implementation.

One remaining issue is how the idle manager discovers which animations are available for a particular model. This is half-automated. All the idle animations in all models have to adhere to a simple naming rule – they have to be named with the string "`idle`" followed by a double digit number. The double digit number has to start at 00 and increase by one for every subsequent animation (animations named with non-consecutive numbers will not be discovered). Then the idle manager will try to enumerate all the animations starting with "idle00" and go as long as the subsequent animations exist. Finally, it will create a table of available animations for every model.

The tables of available animations are created the first time some model requests an idle animation. If the animations are named correctly, the system will discover them all and make sure they are displayed sufficiently randomly. For more information on how this all works please consult the script code.

As discussed before, since the `IdleManager` becomes a global table in the LUA space, it can be accessed from any place in the code. Plus, it conveniently isolates all the operations discussed here so that any intervention in the selection method for animations or their discovery is transparent to its user.

Conversations and conversation management

The complete logic that controls how conversations are performed in Far Cry is written in the form of a few script tables. There is no C++ code related to the conversations. This makes it easier to maintain and completely rewrite. The following text explains how conversational logic works in Far Cry.

Conversations in Far Cry have to be first and foremost – scripted. There must be a conversation script that tells the system how many participants a certain conversation has, and the "flow" of the conversation. Its best explained in an example, so we will look at one conversation from the few that can be found in the Scripts/AI/Packs/Conversations folder.

```
{
Participants = 2,
Script = {
	{
	Actor = 1,
	Duration = 1837,
	SoundData = {
		soundFile = "languages/missiontalk/training/training_generic_H_1.wav",
		Volume = 255,
		min = 10,
		max = 40,
		sound_unscalable = 0,
		},
	},
	{
	Actor = 2,
	Duration = 1420,
	SoundData = {
		soundFile = "languages/missiontalk/training/training_generic_H_2.wav",
		Volume = 255,
		min = 10,
		max = 40,
		sound_unscalable = 0,
		},
	},
	{
	Actor = 1,
	Duration = 1200,
	SoundData = {
		soundFile = "languages/missiontalk/training/training_generic_H_3.wav",
		Volume = 255,
		min = 10,
		max = 40,
		sound_unscalable = 0,
		},
	},
	  },
},
```

This really looks more complicated than it actually is. The table simply describes the lines that will be said in the conversation and the dynamics of the conversation. Each conversation needs to be defined in this way.

In order to know how many people need to join the conversation before it starts, the system needs to know the number of participants in the conversation. That is the first parameter of the conversation script. Most conversations have 2 participants, but the number can be bigger or smaller.

Following is the actual script, separated in lines – every conversation line has to have its own sub table. Every line has to define WHICH one of the actors will say it. This is the property Actor and it can be any number between 1 and the number of participants. This property enables the scripting of normal conversations in which the participants alternate in saying one line each (1,2,1,2,…etc), or monologues (1,1,1,1,.. etc), or rather one-sided conversations (1,1,1,2,1,1,2,1,1,1,2…etc).

Beside the Actor, every script line needs to define the time (in milliseconds) that has to pass until the next script line is executed. This parameter (Duration) gives another way to customize the conversation. If the time duration of one script line is the same or bigger than the time duration of the associated sound file, then the resulting conversation has a very civil dynamic (the next speaker waits until the previous finishes his sentence). But, if this time is shorter, then it is possible to script conversations in which one of the participants appears impatient. This will work naturally since the sounds will be played in 3D sound from the respective positions of the participants – something that cannot necessarily be done by baking-in the whole conversation in a single file.

The final property is the `SoundData` sub-table. This sub table defines al properties of the sound that needs to play when this line is executed – including the actual wav file, the volume, minimum and maximum distance at which the sound will be heard etc.

The lines are executed sequentially in top down.

When a certain number of conversations have been defined in this way, then it is possible to request a conversation with a specific name, or just a random conversation. Even though in Far Cry initially the focus was on having more random conversations, for story development and other reasons it was moved away from that principle. Now the conversations are selected based on their name.

When a conversation is requested, the conversation manager selects one appropriate conversation. It is possible to have multiple conversations under the same name – the system will just select one. When the conversation is selected, the entity (AI object, enemy) that requested the conversation has to invite other entities to join the conversation. He does this by sending a signal to the members of his own group. Whoever receives this signal then joins the conversation by calling the function Join on it. When the amount of joined entities becomes equal to the required participants, the conversation is started. IT IS ALWAYS THE ENTITY WHO ORIGINALLY REQUESTED THE CONVERSATION THAT STARTS WITH THE FIRST LINE.

After a line is selected and the associated sound file is played, the conversation sets a timer with the duration of the selected line. When this timer expires, an event is generated in the conversation that makes it call the function Continue on itself and thus continue the conversation.

When the last line of the conversation is finished, then the participants go back to whatever they were doing previously. The conversation is not repeated until someone else requests it.

To understand the finer details of how conversations are managed and how they are executed on runtime, the actual script logic files for the `AI_ConvManager` and the `AI_Conversation` can be found as LUA files under the same name in `Scripts/AI/Logic`.

The Name generator

The name generator is very simple, and it really can hardly be called a logic script. It just contains a big list of names that were inserted into a table, and a simple function that retrieves the next available name when someone requests it.

To make this possible, an index pointing to the next available name is bumped every time a new name is requested. If the end of the table is reached, it wraps to the beginning.

Signals

CryAISystem provides a powerful, fully customizable tool to make AI entities communicate with each other, that is the Signal system. A signal is an event that can be sent by an agent to another single agent (including itself) or to a sub-set of all the agents currently active in the game. We have already met the concept of signal in the Goalpipe section (see the "signal" goal description for details). In this section, we'll describe:

- how to send signals from an agent's behavior to other agents
- how to define the sub-set of the agents which will receive the sent signal
- how the recipient agent can react to the sent signal

Sending signals

The method to send a signal is the following:

```
AI:Signal(Signal_filter, signal_type, "MySignalName", sender_entity_id);
```

Where:

`Signal_filter`: defines the subset of agents in the game which will receive the signal. It can be chosen among a fixed set of symbols which have the prefix SIGNALFILTER_. The complete list of available signal filters is showed below.

`signal_type`:

1 – the entity receiving the signal will process it only if it's enabled and it's not set to *ignorant* (see `AI:MakePuppetIgnorant()` for details)

0 – the entity receiving the signal will process it if it's not set to *ignorant*
-1 – the entity receiving the signal will process it unconditionally

"MySignalName": the actual identifier of the signal. It can be any non-empty string; for the signal recipient, it must exist a function with the same name either in its current behavior, its default behavior or in the DEFAULT.lua script file in order to react to the received signal.

entity_id: the entity id of the signal's recipient. Usually you may want to put entity.id (or self.id if it's called from the entity and not from its behavior), to send the signal to the sender itself, but you can also put any other id there to send the signal to another entity.

Defining who will receive the signal

With the signal filter, we can define the subset of agents which will receive the signal.

The signal filter parameter in the AI:Signal(...) function call can be one of the following:

Signal filter	The signal is sent to…
0	To the entity specified with the `entity_id` parameter (usually the sender itself but not necessarily)
SIGNALFILTER_LASTOP	the entity's last operation target (if it has one)
SIGNALFILTER_TARGET	the current entity's attention target
SIGNALFILTER_GROUPONLY	all the entities in the sender's group, i.e. the entities with its same group id, in the sender's communication range
SIGNALFILTER_SUPERGROUP	all the entities in the sender's group, i.e. the entities with its same group id, in the whole level
SIGNALFILTER_SPECIESONLY	all the entities of the same sender's species, in the sender's communication range
SIGNALFILTER_SUPERSPECIES	all the entities of the same sender's species, in the whole level
SIGNALFILTER_HALFOFGROUP	half the entities of the sender's group (there's no way to specify which entities)
SIGNALFILTER_NEARESTGROUP	the nearest entity to the sender in its group
SIGNALFILTER_NEARESTINCOMM	The nearest entity to the sender in its group, if it's in its communication range
SIGNALFILTER_ANYONEINCOMM	All the entities in the sender's communication range
SIGNALID_READIBILITY	This is a special kind of signal which is used to make the entity recipient perform a readibility event (sound/animation).

Receiving the signal

The action to be performed once a signal is received is defined in a function like this:

```
MySignalName = function(self, entity, sender)
      …
end
```

where
`self`: the entity's behavior
`entity`: the entity itself
`sender`: the signal's sender

This function is actually a callback which, exactly like the system events, can be defined in the recipient entity's current behaviour, the default idle behavior (if it's not present in current behavior) or in the Scripts/AI/Behaviors/Default.lua script file (if not present in the default idle behaviour).

As for system events, a signal can be used also to make a character change its behaviour; if we add a line like the following in a character file:

```
Behaviour1 = {
      OnPlayerSeen = “Behaviour1”,
      OnEnemyMemory = “Behaviour2”,
      …
      MySignalName = “MyNewBehaviour”,
}
```

This means that if the character is currently in `Behaviour1`, and receives the signal `MySignalName`, after having executed the callback function above, it will then switch its behaviour to `MyNewBehaviour`.

An example

A typical example is when a player’s enemy spots the player: his `OnPlayerSeen` system event is called, and he wants to inform its squad mates, i.e.the guys sharing his group id.
In his default idle behavior (i.e. CoverIdle.lua if the character is Cover), we modify the `OnPlayerSeen` event like this:

```
OnPlayerSeen = function( self, entity, fDistance )
   -- called when the enemy sees a living player

   AI:Signal(SIGNALFILTER_GROUPONLY, 1, "PLAYER_SPOTTED",entity.id);

end,
```

Here we have defined a new signal called `PLAYER_SPOTTED`.
The next step is to define the callback function. Let’s assume the other members in the group have the same character, we then add the callback function to the same idle behaviour in which we have just modified `OnPlayerSeen`.

```
PLAYER_SPOTTED = function (self, entity, sender)

   entity:Readibility("FIRST_HOSTILE_CONTACT");
   entity:InsertSubpipe(0,"DRAW_GUN");

End,
```

This will make the guys, including the signal sender itself, who has the same behaviour, change their animation and produce some kind of “alert” sound (readibility), and then draw their gun. Notice that by modifying its idle behavior, we

create a *default* callback which will be executed for any behaviour the character is in. Later on, we may want to override this callback in other behaviors. For example, if we wanted the character to react differently whether its in idle or attack behaviour, we'll add the following callback function in the CoverAttack.lua file:

```
PLAYER_SPOTTED = function (self, entity, sender)

   entity:SelectPipe(0,"cover_pindown");

End,
```

where "cover_pindown" is a goalpipe that makes the guy hide behind the nearest cover place to the target.
We can extend this to other characters: if there are group members with different characters, e.g. Scout, Rear etc., and we want them to react as well, we must add the PLAYER_SPOTTED callback to their idle/attack behaviour as well.

Finally, we want the guys to switch their behaviour from idle to attack if they see the player. We'll then add the following line to the character (Scripts/AI/Characters/Personalities/Cover.lua in the example):

```
CoverIdle = {
   …
   PLAYER_SPOTTED = "CoverAttack",
},
```


Part VI – Tips and tricks

Since the most important thing in designing a level is its appearance to the player playing it, it is beneficial to understand the basic principles in conveying a situation to the player (in accordance with the game rules). The game rules of the game Far Cry have been laid out and AI-wise made into several classes of characters that behave differently from one another and BASED ON THE ENVIRONMENT in which they are placed.

Thus the responsibility of the level design is as important as the fact that the behaviors must be consistent. Dry and repetitive design will surely present a dry and repetitive game experience to the player, no matter what features are present in the behavior.

In this document, the basic building blocks of a Far Cry level will be discussed from a point of view of functionality in the AI characters defined in the game Far Cry. Using these building blocks, and combining them into larger units, it is conceivable that a pleasant game-play experience can be created.

Cover objects

Cover objects in the game should be plentiful, and when only they are visible – they should show the conceptual (logical) layout of the map. They should show areas where AI would be expected to move, lines of advancement and more importantly they should show a clear separation between action bubbles.

There is a way of using cover objects badly, which is when they are placed in clearly unreachable places, or on steep inclines, or on edges of action bubbles (or cliffs). All of these occurrences will potentially break the consistency of the enemy behavior, and should be used sparingly and with forethought.

As a general conclusion, more cover is always better than less. Having more cover places no speed penalty on the execution, and only a symbolic increase in memory consumption.

A cover spot does not necessarily have to correspond to an actual object. A fake cover spot can be created to facilitate a movement objective for the AI where for artistic (or other reasons) there is not a real object. The enemy will recognize such a cover spot transparently and act accordingly.

Cover objects can also be of half-height. An enemy choosing such a hide spot will adjust his body stance automatically. This is done transparently.

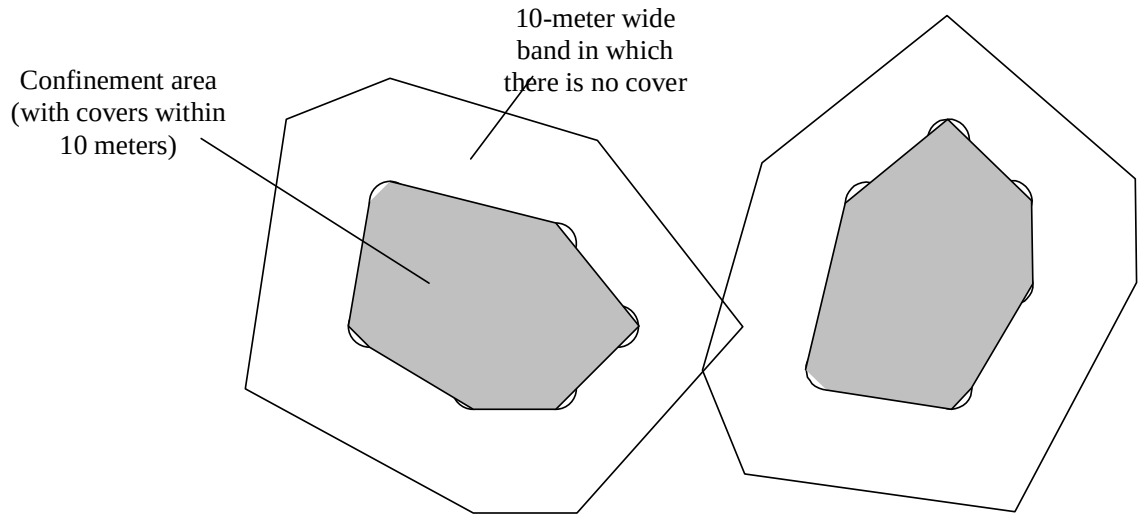
When an object is placed to serve as a cover object OUTDOORS, it should generally have a radius no larger than 3 meters. While this is not mandatory, it will look better for the movement of the enemies. If a larger cover than 3 meters is needed, then it should be broken up into smaller pieces, or declared as non-hide able, and then embellished with fake hiding objects. For INDOORS this is not valid, as hide point processing is completely different.

Cover is used in some instances as ordinary movement targets (during idles and so on). The pig for example uses the cover as anchors to move trough the environment. This is why cover placing is crucial for making a living world.

The (dreaded) "10 meter cover" rule

This is the very basic OUTDOOR rule for the movement of the enemies. You can define areas within which the enemy will be able to move freely – in combat and in idle. These areas should be defined as areas in which there is cover objects separated by no more than 10 meters. Cover objects can be placed closer than 10 meters and there can be an arbitrary number of them.

An enemy placed in such an area will move freely around in it, and use the cover object in combat. If this area is surrounded with a 10m long band (along its edges) in which there is no cover object, then an enemy placed within this area WILL BE CONFINED TO IT. This is illustrated in the following picture.



This is a way to make sure that the player encounters a particular enemy at a particular place. The combat in this area will depend solely on the player's movements, and it will be different every time, but it will ALWAYS happen here.

These areas should be viewed as tools to localize enemy movements, but also as tools to create a certain situation (based on the intrinsic behavior of the enemies). This is discussed in the following section.

Simulating specific behavior with smart cover placement

So we have established that the distribution of cover (within 10m) dictates the movement pattern of the enemies (to a point). Another thing that affects the movement of the enemies is their intrinsic behavior pattern. For example, a cover character will advance using cover to the position of the player, while a scout will stay at approx 15-20 meters from the player, and try flanking him left and right. Knowing these properties is crucial to being able to take advantage of them – combined with the 10-meter rule.

The most basic cover structure is of course the **single cover** object. Pretty much all enemies when placed in a situation like this will stick to this object and periodically come out to engage the player. This kind of setup is useful when you want the enemy to protect a certain spot or path of movement. He will use this cover in all directions according to the player's movement, so the surrounding environment should work toward guiding the player toward this spot (otherwise the player can just circumvent the setup).

Next, cover can be arranged in a **straight line** (with cover at least 10 meters from each other). This is the first setup in which differences between personalities become obvious. Let's say that this cover line is placed perpendicular to the player's approach vector. If you place a cover here, he will not be able to approach the player using cover, since the only cover will now be to the left and right of him. Thus, he will hang around his initial cover spot and fight from there (behaving like a single cover character). However, if you place a scout in this setup, then he will move left and right along the line and protect the full length of the line. Since he will not have cover behind him, he will be forced to stay and protect the line until he is dead.

The line has other cool properties. For example, if the player now relocates and moves in an approach vector parallel to the line, then the cover will now be able to approach the player using the line cover object. In this situation the cover will behave better then the single cover situation. A rear mercenary (on the other hand) will in both situations generally stick to his cover (like a single cover object) and only flank rarely and reluctantly.

So there are a lot of situations in which a line would be an ideal choice (protecting a path, protecting the perimeter of a compound, fighting in a thin ravine etc). Since the enemy behavior will depend on the orientation of the line from the players approach vector, a variety of combinations will emerge.

The next in the hierarchy of cover object organization is of course the **curved line**. Curved lines exhibit the same properties as the straight line, except they provide more freedom in defining the movement of the enemy. Some cool flank situations can be created using an arc around the player's presumed position. Using scouts you can give the player the impression that the enemies are organizing themselves to surround the player. Placing a few covers (or rears) in the middle will create the illusion of support fire while the scouts flank. The shape of the curved line can be made to best fit the surrounding environment. As the player changes position, so the roles of the enemies will become more or less dominant in the combat.

Of course, cool situations can be produced with **combining and intersecting lines**. In this way you can provide advance cover lines for covers, and flank lines for scouts. They will both use all cover lines, so as the player relocates the combat will spread accordingly, but along these battle lines that you define.

You are encouraged to try all these situations in a test map and watch the emergent behavior that will happen based on the simple rules of movement for the different classes of enemies.

Next step up is to take the cover arrangement into 2 dimensions – **areas**. Like already shown in the picture, areas can have any shape, but it is imperative that they are created with any sort of idea. Placing random cover objects on the map will create SOME distribution and some areas, but this will result in a much averaged experience and will not yield the results that can be achieved with careful placement.

Using soft cover in a smart way

Far Cry has a feature called soft cover. This represents cover that enemies (and the player) can pass trough, but CANNOT see trough. Soft cover is one feature that has been not used to its fullest extent because of the double nature that it has.

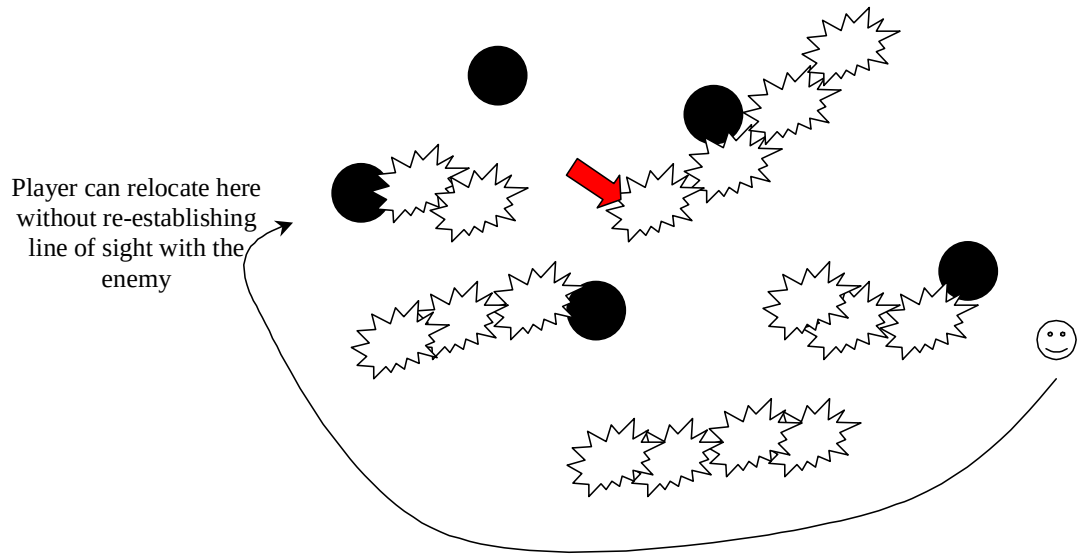
Soft cover occludes the visual field of the player. When used in a bad way it introduces frustration and generally is counter productive. Here are a few guidelines on how to maximize the usefulness of soft cover.

Soft cover objects should never be as high as the player (or higher). This means that at some position they will occlude the player's vision and this is bad. Worse however is that any enemy standing behind soft cover like that will be invisible to the player. Even worse is that the enemy will not be able to see the player and so will not be able to respond correctly. So as a general guideline, head high bushes (soft cover objects) should not exist in the game WITHIN THE ACTIVE PORTION OF THE ACTION BUBBLE.

The ideal height of soft cover objects is such that the player can hide behind them IF HE CROUCHES. This makes certain that the player made a conscious decision do break off visual contact with the enemies (he has to press crouch). It also does not hide an enemy standing behind this soft cover object. More importantly, it gives very strong player language that crouching can help you hide and relocate. Some soft cover can be prone height, but this is impractical since the prone movement speed is drastically lower and the player cannot consider moving in prone trough cover as an advantage. However, for certain situations this could be an added plus.

The player should be able to relocate when he is under fire using PRIMARILY SOFT COVER. This is because it is unnatural to place hard cover next to each other (thus providing an escape route during which the enemy will not be able to establish a line of sight), but its very natural to do it using soft cover. So the player will know to duck and then relocate, and likely attack the enemies from the side or from the back. This is a way of giving the player the upper hand in flanking (which is impossible in the current layout of the maps).

The following picture illustrates this method of soft cover placement.



All of the soft cover in the picture is half the player height, so it does not impair the vision of the player. The player can relocate along the shown line in crouch mode while the enemies try to advance to the last known position of the player. This will enable PLAYER flanking in our game – an important player tool.

Another smart usage of soft cover is when you want to provide a sneak way to an action bubble. If the approach is on an inclined position (a hill for example), several "prone height" objects can be placed on the hill encouraging the player to prone and approach. On the other hand, placing low prone soft cover in a valley where the enemy has an inclined position is not advisable.

Appendix A – Atomic goal reference

This is a list of all atomic commands currently supported in the Far Cry CryAISystem.

"acqtarget" – This goal can be used when the entity that executes it needs to get a specific target as its attention target – like for example a tag point. The target that will be acquired has to be specified by its name, and name selection will work ONLY for tag points, NOT objects of any other type. However, IF the string parameter is an empty string, then the acquired target will be the last operation target. You can find out more about the last operation target here.

Parameters: one string containing the name of the tag point that needs to be acquired as a target, or an empty string to acquire the last operation target.

Notes: While you can make this goal blocking, it always executes in a single frame, so it will not block anything.

Examples:

```
AI:PushGoal("close_attack","acqtarget",0,"");
      -- acquire LastOpTarget
AI:PushGoal("retreat","acqtarget",0,"safe_point");
      -- acquire Tag Point calles “safe_point”
```

"approach", "backoff"– These directives make the entity that uses them approach / (back off from) its attention target to a certain distance, specified as a parameter. This goal will execute in environments where there are obstacles and will avoid them automatically. The goal will be automatically skipped if a path close enough to the attention target cannot be generated.

Parameters: Desired distance from the target in meters.

Notes: The approach directive has an interesting twist – if the distance is less than 1 meter, then it is no longer the distance, but a percentage of the initial distance from the target. So for example if you specify 0.5 that means the enemy will approach 50% closer to his target. Another optional parameter for the APPROACH goal is a fifth parameter (that can be 1 or 0). If this parameter

is 1, then the goal will use the LAST OPERATION TARGET as the target to approach to.

Examples:

```
AI:PushGoal("scout_hunt_run","approach",1,0.5);
      -- approach 50% to your target
AI:PushGoal("scout_comeout","approach",1,0.9,1);
      -- approach 90% to your last operation target
AI:PushGoal("scout_form","approach",1,2);
      -- approach 2 meters to target
```

"bodypos" – Controls the body stance of the enemy executing it. This is just a number to the AI system. You are free to pass any number to this goal and that number will be redirected to the proxy which can adapt the body stance. After the stance is set, all subsequent operations will operate on the new stance. The parameters are only symbolic constants defined in the game Far Cry.

Parameters: Desired stance, one of the following symbolic constants: BODYPOS_STAND – combat standing; BODYPOS_CROUCH – combat crouching; BODYPOS_PRONE – combat prone; BODYPOS_RELAX – relaxed standing; BODYPOS_STEALTH – stealth standing .

Notes: The stance does not just affect the way an entity stands, it also affects its movement speed, type of animation and whether or not it has the ability to run.

Examples:

```
AI:PushGoal("retreat_back","bodypos",0,BODYPOS_STAND);
      -- goes into combat stand stance
```

"bodypos" (For ground vehicles) – Tells the ground vehicle if it must update or not its path while moving. Update path while moving is required for example if the vehicle must chase/follow a moving target.

Parameters: 0 – Do not update path; 1 – Update path

"bodypos" (For helicopters) – Tells the helicopter if it must ignore or not the possible collision with trees.

Parameters: 1 – Do not ignore collisions with trees; 1 – Ignore collisions

"firecmd" – Usually, when an AI object has its target within attack range, it will fire at it. Firing happens in short bursts if the target is on the far end of the attack range, or continuously if the target is close. This command enables or disables this automatic processing, by telling the AI object whether it is allowed to fire or not. This command does not provoke a single shot; it just generally controls the ability to fire.

Parameters: 0- Do not fire, 1- Fire at will, 2 – Fire randomly

Notes: Allowing the enemy to fire at will does not mean that it will automatically start firing at nothing. It will fire only when there is a live target to shoot at. On the other hand, allowing the enemy to fire randomly means allowing him to fire at any time and at anything – even thin air ☺

Examples:

```
AI:PushGoal("shoot_the_beacon","firecmd",0,2);
      -- shoots wildly at anything
AI:PushGoal("scared_shoot","firecmd",1,1);
      -- shoot when target in range
AI:PushGoal("retreat_back","firecmd",0,0);
      -- do not shoot under any circumstances
```

"run" – A toggle switch for the movement speed of the enemy. When the enemy is switched into run mode (by specifying parameter 1) then all subsequent operations will be performed while running. This will change only if another run goal is executed and it switches the enemy back into walking.

Parameters: 0 – not running, 1- running.

Examples:

```
AI:PushGoal("minimize_exposure","run",0,1);
      -- start running
```

"jump" – Will cause the AI object executing it to jump. This goal will calculate the jump parameters based on the method of target selection, distance etc. The goal will execute until the object is at the jump position. This goal executes under the constraints of the physics system, so some small deviation from the jump position is possible.

Parameters: *Parameter 1*: Method of jump selection. This is one of the symbolic constants used for selecting a hide object (like HM_NEAREST). Consult the reference to the hide goal for a full description of these methods. This parameter can be 0 which will mean that the object will jump to his LAST OPERATION TARGET.

Parameter 2: Distance in which to search for a jump target in the selected method. The distance is given in meters. If the previous parameter was 0, this parameter is ignored and should also be set to 0.

Parameter 3: Calculation/Execution flag. If this is set to 1, the jump will be only calculated and the structures necessary will be initialized with the jump data – but the actual jump will not be executed. If this parameter is 0, then the jump is calculated and executed.

Parameter 4: (Optional, defaults to 45). This parameter defines the angle of jump from the horizontal plane. A higher angle means a higher jump.

Examples:

```
AI:PushGoal("jump_to","jump",1,0,0,0,self.Properties.fJumpAngle);
            -- actual jump executed here
AI:PushGoal("jump_left_hide","jump",1,20,HM_LEFTMOST_FROM_TARGET,1);
            -- calculate jump only DONT ACTUALLY JUMP
```

"timeout" – This goal provides a way to stop the execution of a pipe for some time period (used in combination with the blocking flag).

Parameters: First parameter is time in seconds. Optionally you can specify a second parameter (also time in seconds) and the actual timeout will be a value randomly selected between the two variables.

Examples:

```
AI:PushGoal("scout_find_threat","timeout",1,1,2);
      -- wait between 1 and 2 seconds
AI:PushGoal("scout_firesome","timeout",1,1);
      -- wait 1 second
```


"locate" – This goal stores a specific target and in the last operation result. Using this goal the AI object can locate ANY other AI Object including tag points, the player, formation points, beacons etc. Depending on the parameters to the locate goal, a different kind of target is stored in the last operation result.

Parameters: The name of the AI Object, or point to locate. There are some special strings to locate special game items: "**player**" locates the human player in single player game; "**formation**" locates the nearest formation point (if a formation for the group that this AI Object belongs to exists); "**atttarget**" stores the attention target of the AI Object into the last operation result; "**hidepoint**" locates a random hide point in a 30 meters radius from the objects current position; "**beacon**" locates the beacon of the group that the AI Object belongs to.

The argument does not have to be necessarily a string – it can be a type of an object that is requested. Then, the nearest object of a matching type will be stored in the last operation result.

Examples:

```
AI:PushGoal("rush_player","locate",0,"player");
      -- stores the human player into the LOR (last op result)
AI:PushGoal("run_to_beacon","locate",0,"beacon");
      -- stores the beacon of the group in the LOP
AI:PushGoal("get_toolbox","locate",1,AIAnchor.AIANCHOR_TOOLBOX);
-- stores the nearest object of type AIAnchor.AIANCHOR_TOOLBOX
```

"pathfind" – This goal generates a path from the current position of the AI object to a desired position specified as a parameter to the goal. It is recommended that this goal be blocking at all times, because the pathfinder might take some time to generate the path. That is not a requirement however, and you are free to have the AI Object do something else while pathfinding, but make sure you use that path only after pathfinding has finished. The usefulness of this goal is questionable since all movement goals generate paths when they execute, but its there for more control over where the path is generated to.

Parameters: string that denotes the name of an existing AI object in the map, or an empty string if last operation result should be used as the target for the operation.

Examples:

```
AI:PushGoal("get_gun","pathfind",1,"");
      -- generates a path to the last operation result
AI:PushGoal("cargo_run","pathfind",1,"cargo_run");
      -- generates a path to the object named “cargo_run”
```

"trace" – This goal makes the agent move along the path generated in a previous call to pathfind. The agent will start moving on this path, but will still evaluate threats along the way and will keep his existing target in focus.

Parameters: There are 2 optional parameters for this goal. The first parameter, if it is specified as 1, then the object will look in the direction where he is moving while he is in motion. If this parameter is 0, then the object will focus on its attention target at all times. The second parameter, if it is non zero, will enable a single-step execution of the generated path – basically the AI object will advance to the next path point and the goal will finish. The whole generated path can be traced later by subsequent calls to trace.

Examples:

```
AI:PushGoal("get_gun","trace",1,1);
      -- trace while looking where you are going
AI:PushGoal("comeout","trace",1,0,1);
      -- trace while looking at your target but only one
step
```

"signal" – This goal is used to signal some event to the script that controls the agent. This signal is user defined and is not related to the AI system at all. The AI system is only responsible for propagating this signal to the recipients specified through a filter parameter. There is a separate discussion on signaling and the processing of signals in this manual.

Parameters: This goal has three parameters that need to be defined, and they are as follows: The first parameter is the signal priority. To understand how to use this parameter, please consult the documentation of signalling in this manual. The second parameter is the actual signal text that needs to be sent, and the last parameter is the filter which needs to be used in order to determine who will receive this signal. There is a complete list of all supported signal filters at the end of this appendix.

Examples:

```
AI:PushGoal("seek_target","signal",0,1,"LOOK_FOR_TARGET",0);
      -- signal at normal priority to yourself
AI:PushGoal("delay_headsup","signal",1,1,"wakeup",SIGNALFILTER_SUPERGROUP);
  -- send a normal priority signal to everyone in your group
```

"ignoreall" – This goal makes the agent stop evaluating threats and acquiring new targets. This is useful if the scripter wants to achieve a very specific behaviour for a very specific AI object and doesn't want it to be distracted by any other targets. The AI object WILL STILL receive signals however. There is a separate function to make the object also irresponsive to signals.

Parameters: 0 – stop ignoring all; 1- ignore all;

Notes: This goal should be used with caution as it can create a situation where the AI object does not receive any sensory input. To a player this appears as and AI object that is stuck and will not react. USE WITH CAUTION.

Examples:

```
AI:PushGoal("delay_headsup","ignoreall",0,0);
        -- stop ignoring all targets
```

"strafe" – This goal makes the agent strafe around his attention target. This directive has been put in as a convenience for scripters that wish to script an agent that plays more like a real human opponent in a multiplayer game. Notice that the agent strafes automatically when tracing out a path, so this directive should really be used for scripting close combat maneuvers.

Parameters: Amount of meters to strafe from current position. Positive value means strafing right while negative value means strafing left.

Notes: While strafing the AI object will take care not to cross any forbidden areas and navigation modifiers, but will not generate a path to strafe. This is an important distinction.

Examples:

```
AI:PushGoal("crowe_dodge_1","strafe",1,-5);
      -- strafe 5 meters to the left.
```

"strafe" (for vehicles) – Activate/deactivate brakes for vehicle.

Parameters: 0: release brakes; 1: activate brakes.

"lookaround" – Using this goal, you can make the AI object look in a random direction around him, make him look like he is looking around. The object will pick a random look point around him and turn to it.

Parameters: A value defining the maximum allowed turn angle in radians (approximately 0 -7). The actual selected angle will be a random number between 0 and the specified number.

Examples:

```
AI:PushGoal("investigatesound","lookaround",1,1);
-- look in direction in the range 0 to 1 radians from
current orientation
```

"lookat" – This is an extension to the lookaround goal, in that you can force the AI object to look in a certain direction, or force him choose between a range of degrees around him. The AO object will look in the desired direction. The angles you give are relative to the current viewing direction of the object (so for example, specifying 180 for start angle and end angle will cause the object to look directly behind him).

Parameters: Start and end angle – the object will choose a random value between these values (in degrees). If they are the same, then you can make the object look in any specific direction.

Examples:

```
AI:PushGoal("hold_spot","lookat",1,-45,45);
      -- look left right in the forward direction
AI:PushGoal("special_hold_position","lookat",1,-90,90);
      -- look left right but not back
```

"hide" – This directive is fairly complex. It will cause the puppet to look for a hiding spot based on the parameters that you supply. After it has found it (this happens instantaneously) it will move toward it. It is considered executed when the puppet has arrived at the hiding spot. If a hiding spot with the desired parameters is not found, a signal is generated to the puppet (NoHidingPlace, NoNearerHidingPlace). **Parameters**: distance in meters how far a hide spot should be found (if there is no hide place in this distance, the signal is generated). Next is the hide method which can be: HM_NEAREST, HM_NEAREST_TO_TARGET, HM_FARTHEST_FROM_TARGET, HM_NEAREST_TO_GROUP, HM_FARTHEST_FROM_GROUP, HM_NEAREST_TO_LASTOPRESULT, HM_FARTHEST_FROM_LASTOPRESULT HM_LEFTMOST_FROM_TARGET, HM_RIGHTMOST_FROM_TARGET. This list can be extended.

1. **"form" –** This directive will instruct the puppet to create a formation for its group. The formation must be identified with a name, and it has to already exist (it is most likely defined in the editor). Members of the puppet's group can then request locations of formation points. Parameters: string defining the name of the formation. If this is a zero string ("") then any existing formations are cancelled.

2. **"branch" –** This directive should be used when you know what you are doing with pipes. If you use a non blocking goal that you know will take some time to execute, but you don't know how long, you can use this directive. It will check whether all previous directives have finished, and if so, it will let the pipe continue executing. If not, it will jump back as many directives as you have specified as a parameter, and resume execution from there. BE CAREFUL! This is a mistake:

   approach 5 nonblock
   branch –1

   You see, here you will keep adding approach directives every time the branch is executed. This means this loop will never exit. This is better

   approach 5 nonblock
   timeout 0.1 block
   branch –1

   You can put as much directives between your non blocking directive and the branch command, like in this example:

   approach 5 nonblock
   lookaround block
   timeout 0.5 block
   lookaround block
   signal "cough"
   timeout 0.5 block
   branch –5
   firecmd 1

   This will make the puppet lookaround an cough while it is approaching the target, and only start firing when it has actually approached to 5 meters of the target. **Parameters**: how many directives to jump over.(NOTE: theoretically, it is possible to make a jump forward – a positive value. I did not remove this because someone might find a use for it. Bear in mind that jumping back beyond the start of the pipe or forward pass the end of the pipe WILL CAUSE UNDEFINED BEHAVIOR.)

3. **"Nesting GoalPipes"** - it is possible to change from the current pipe to another, process the new pipe and *return to the original pipe*; this is also referred to as 'nesting' a GoalPipe inside another.
4. **"devalue"**- this directive devalues the current attention target of

the enemy. To devalue means that it stops being the attention target, plus it can no longer acquire it NORMALLY for some period of time. HOWEVER, such targets can be freely acquired by an explicit call to acqtarget. This will also not work for players nor other enemy targets – it only works for waypoints!!

5. **"forget"** - Sometimes the memory of a target is very long, even though you have done seemingly everything to find it, you couldn't. In these cases, it is convenient to instruct the enemy to forget this particular memory. So, if your current target is a memory target (which you would know because you would receive a callback like OnEnemyMemory), this directive will make the enemy instantly forget the memory target.
6. **"clear" –** this directive is used to clear the state of the enemy. Any goals executing will be erased. The enemy will release his attention target. The only thing that remains after a clear is the current pipe. So, it is possible to push more goals after a clear – they are guaranteed to be executed in a clean environment.

Appendix B – Lua script functions reference

AI system script functions

AI system script functions are lua functions defined in the AI system and owned by the common script object named AI, which is the reference to CryAISystem inside Lua.

For example, to call the function `GetAttentionTargetOf()`, you'll write

```
Target = AI:GetAttentionTargetOf(self.id);
```

AI:GetAttentionTargetOf(entity_id)

Retrieve the attention target of an entity

Parameters:

(number) `entity_id`: the entity's id

Returns:

- (table) entity – the reference to the attention target, if it is an entity
- (number) `AIOBJECT_DUMMY` – if the target is a dummy AI Object (waypoint, beacon etc)
- (number) `AIOBJECT_NONE` – if the entity has no attention target
- `nil` – if the passed `entity_id` parameter doesn't correspond to an active entity in the game

`AI:FindObjectOfType``(search_center, range, object_type)`

Find a game object of a given tipe, in the given range around the given entity. If there are more objects of that type, there's no rule to determine which one of them will be returned.

Parameters:

- (number/vector) `search_center`: the center point of the search. The objects will be searched around this point; the parameter can be either an entity id (number) or point coordinates (vector {x,y,z})
- (number) `range`: the radius of the search (in meters)

(number) `object_type`: the type of the object we want to find. See `AI:RegisterWithAI(..)` for a list of available types. Note: to search for a vehicle, only `AIOBJECT_VEHICLE` can be used in this function. There's no distinction between cars, boat and helicopters and using any of the types `AIOBJECT_CAR`, `AIOBJECT_BOAT` and `AIOBJECT_HELICOPTER` will produce no result.

Returns:

- (string) `object_name`: name of the object if it's found
- `nil`: if no object is found

See also:

`AI:RegisterWithAI(..)`

`AI:RegisterWithAI``(entity_id, object_type [, `*`properties`*`][, `*`properties_Instance`*`])`

Tells the AI System to register an entity with AI. It basically assigns an AI agent of a given type to an entity.

Parameters:

- (number) `entity_id`: the entity's id
- (number) `object_type`: the type of the object's AI. It can be either an anchor type or one of the following:

`AIOBJECT_PUPPET`	An entity which has a living being-like AI (mercs, animals etc). This kind of entity has usually the physics of a walking creature, it has a character and behaviours, it's sensitive to the player's presence, it can acquire targets and receives system events like OnPlayerSeen etc. It can execute goalpipes. It is usually registered with the entity class "Player" in the file ClassRegistry.lua

`AIOBJECT_CAR/`
`AIOBJECT_BOAT/`
`AIOBJECT_HELICOPTER`
	An entity which has a vehicle-type AI and physics (cars, boats, helicopters etc). This kind of entity has a character and behaviours, it is sensitive to the player's presence, it can acquire targets and receives system events like OnPlayerSeen etc. It can execute goalpipes, and it is able to move on its own like an `AIOBJECT_PUPPET`. It is usually registered with the entity class "Vehicle" in the file ClassRegistry.lua

`AIOBJECT_WAYPOINT`	A waypoint
`AIOBJECT_ATTRIBUTE`	An attribute object (see attributes)
`AIOBJECT_MOUNTEDWEAPON`	A mounted weapon in a vehicle or on ground
`AIOBJECT_PLAYER`	The player
`AIOBJECT_DUMMY`	A dummy AI object (i.e. beacon) it's not usually used for registering an AI, but only to retrieve the attention target of an entity
`AIOBJECT_SNDSUPRESSOR`	A sound suppressor object (see Scripts/Default./Entities/AI/SoundSupressor.lua and Scripts/Default./Entities/Others/TV.lua for details)
`AIOBJECT_NONE`	An object with no AI

- `properties`: (optional) for puppets and vehicles, it's a table and it defines the set Properties (of the entity instance) which can be modified in the editor.
- `propertiesInstance`: (optional) same as `properties`, it defines the set Properties2 in the editor.

```
AI:GetGroupCount(entity_id)
```

Retrieve the number of members in the given entity's group

Parameters:
(number) entity_id: the entity's id

Returns:
(number) number of members in the entity's group

```
AI:GetGroupOf(entity_id)
```

Retrieve the group ID of an entity

Parameters:
(number) entity_id: the entity's id

Returns:
(number) the group ID of the entity

See also:
entity:**ChangeAIParameter**(..)

```
AI:MakePuppetIgnorant(entity_id, bIgnorant)
```

Turns on/off an entity's perception system (if it's a puppet or a vehicle). If it's set to "ignorant" by passing it 1, an entity will not be able to acquire targets, nor to receive system events, nor signals (unless it's sent with signal_type = -1; see Signals for details).

Parameters:
(number) entity_id: the entity's id

(number) bIgnorant: 1 – turn the entity ignorant (perception system off); 0 - turn the entity non-ignorant (perception system on)

Example of usage

A typical situation is when you want to force a guy performing a certain action – for example "run to a vehicle" – without being influenced by the player presence: if you don't set the guy *ignorant*, during his run to the vehicle he can possibly acquire the player as a target, performing the OnPlayerSeen event and switch behavior, forgetting about going to the vehicle.

```
AI:GetPerception()
```

Returns the value of player's Perception coefficient (float number from 0 to >10). When the perception coefficient is 10 (or sligthly above), enemy is seeing the player. You can use intermediate threshold level (for example 7) to make the enemy switch to different "intermediate" states like "interested", "alerted" and so on.

```
AI:IsMoving(entity_id)
```

Tell if an entity is moving

Parameters:
(number) `entity_id`: the entity's id

Returns:
- 1 if the entity is moving
- 0 otherwise

```
AI:Signal(signal_filter, signal_type, signal_name, entity_id)
```

Sends a signal to an entity or a group of entities (See Signals chapter for a full coverage of how to use signals)

Parameters:
- `signal_filter`: defines the subset of agents in the game which will receive the signal. The signal filter can be one of the following:

Signal filter	The signal is sent to...
0	To the entity specified with the `entity_id` parameter (usually the sender itself but not necessarily)
SIGNALFILTER_LASTOP	the entity's last operation target (if it has one)
SIGNALFILTER_TARGET	the current entity's attention target
SIGNALFILTER_GROUPONLY	all the entities in the sender's group, i.e. the entities with its same group id, in the sender's communication range
SIGNALFILTER_SUPERGROUP	all the entities in the sender's group, i.e. the entities with its same group id, in the whole level
SIGNALFILTER_SPECIESONLY	all the entities of the same sender's species, in the sender's communication range
SIGNALFILTER_SUPERSPECIES	all the entities of the same sender's species, in the whole level
SIGNALFILTER_HALFOFGROUP	half the entities of the sender's group (there's no way to specify which entities)
SIGNALFILTER_NEARESTGROUP	the nearest entity to the sender in its group
SIGNALFILTER_NEARESTINCOMM	The nearest entity to the sender in its group, if it's in its communication range
SIGNALFILTER_ANYONEINCOMM	All the entities in the sender's communication range
SIGNALID_READIBILITY	This is a special kind of signal which is used to make the entity recipient perform a readibility event (sound/animation).

- `signal_type`:

  1 – the entity receiving the signal will process it only if it's enabled and it's not set to *ignorant* (see `AI:MakePuppetIgnorant()` for details)

  0 – the entity receiving the signal will process it if it's not set to *ignorant*

  -1 – the entity receiving the signal will process it unconditionally

- `signal_name`: the actual identifier of the signal. It can be any non-empty string; for the signal recipient, it must exist a function with the same name either in its current behavior, its default behavior or in the DEFAULT.lua script file in order to react to the received signal

- `entity_id`: the entity id of the signal's recipient. Usually you may want to put entity.id (or self.id if it's called from the entity and not from its behavior), to send the

signal to the sender itself, but you can also put any other id there to send the signal to another entity (using 0 as signal_filter).

Example:

```
AI:Signal(SIGNALFILTER_GROUPONLY, 1, "PLAYER_SPOTTED",entity.id);
```

AI:FreeSignal(signal_type, signal_name, position, radius)

Sends a signal to every puppet/vehicle inside a circular zone centered in a given point in the map (See Signals chapter for a full coverage of how to use signals). The signal is *anonymous*, that is the receiver will not know who is sending it (the sender parameter in its callback function will be `nil`).

Parameters:

- `signal_type`: same as AI:Signal()
- `signal_name`: same as AI:Signal()
- `position`: a point in the vector format {x,y,z} which specifies the center of the circular zone
- `radius`: a point in the vector format {x,y,z} which specifies the radius of the circular zone

Example:

Suppose you're writing the behaviour of a flashbang grenade, and specifically the event in which the grenade explodes. You want to inform every entity in a radius of 30 meters, no matter who they are and whose side they're on. You may want to use the `AI:FreeSignal()` function for that:

```
AI:FreeSignal(1, " FLASHBANG_GRENADE_EFFECT",entity:GetPos(),30);
```

Then, in the behaviors of the entities you want to be affectable by a flashbang grenade, you add the callback function:

```
FLASHBANG_GRENADE_EFFECT = function(self, entity, sender)
   -- make the entity lose his sensorial capabilities for a while
   entity:InsertSubpipe(0,"stunned");
   -- make it visible to the player with animation/sound
   entity:Readibility("FLASHBANG_GRENADE_EFFECT",1);
end,
```


AI:SetTheSkip(entity_id)

Set an entity to be skipped in raytrace tests (for sighting etc). Only one entity can be set to be skipped at a time. This entity will become "transparent" to sighting tests.

Parameters:

- signal_type: the id of the entity to be set as skipped. If 0, no entity is set

AI:Cloak(entity_id),
AI:DeCloak(entity_id)

These functions are now deprecated and actually uneffective.

AI:SetSpeciesThreatMultiplier(species_id,multiplier)

Set the "threat multiplier" coefficient for a species. It's used to determine the threat level of an attention target, and it's a property shared by all the entities belonging to that species.

Parameters:

- species_id: the species' id
- multiplier: a float number between 0 and 1. The higher it is, the more "threatening" is the species and the more prioritary will become an attention target of that species.

AI:EnablePuppetMovement(entity_id, bEnable [,*duration*])

Enable/disable an entity (whose AI is of type puppet/vehicle) for movement, optionally for a given duration. If it's disabled, the entity cannot move.

Parameters:

- entity_id: the entity's id
- bEnable: 1 – the entity can move; 0 – the entity is not enabled to move

- `duration`: (optional) the time in ms for which the entity is enabled to move. It's uneffective if `bEnable=0`.

`AI:GetStats``(stats_table)`

Fills a table with statistics on the currently played level.

Parameters:

- `stats_table`: a lua table. You may want to use an empty table (i.e. statsTable = {};) and make it fill by the function. Notice that this is an *output* parameter. The table will then be filled in with the following fields:

`AVGEnemyLifetime`	Average life time of player's enemies
`AVGPlayerLifetime`	Average life time of player
`CheckpointsHit`	Number of reached checkpoints in the level
`EnemiesKilled`	Number of enemies killed by the player
`ShotsFired`	Number of shots fired by the player
`ShotsHit`	Number of successful shots fired by the player
`SilentKills`	Number of idle/inactive enemies killed by the player
`TotalEnemiesInLevel`	Total number of enemies in the level
`TotalPlayerDeaths`	Total number of player's deaths
`VehiclesDestroyed`	Number of vehicles destroyed
`TotalTimeSecs`	Level duration in seconds
`TotalTime`	Level duration in (string) time format "hr:min:ss"

Example:


```
local stats = {};

AI:GetStats(stats);

System:Log(sprintf("Number of enemies killed: %d"),
stats.EnemiesKilled);

System:Log(sprintf("Total number of enemies: %d",
stats.TotalEnemiesInLevel));

System:Log(sprintf("Kill accuracy: %d", stats.EnemiesKilled /
stats.TotalEnemiesInLevel));
```

AI:GetAnchor(entity_id, anchor_type, radius)

Retrieve the closest anchor of a given type to an entity.

Parameters:

- (number) entity_id: the entity's id
- (number) anchor_type: the type of anchor to search (see Anchors)
- (number) radius: the maximum distance from the entity to search the anchor within.

Returns:

- The name of the ancor if it's found
- (nil) if the anchor is not found

Example

In the following example, we want a sniper to find an anchor that the level designer has placed in the level to define a sniper shooting spot for the AI. If the anchor is found, the sniper will go there:

```
local anchorName = AI:GetAnchor(entity.id, AIAnchor.AIANCHOR_SHOOTSPOT,
10);

if (anchorName) then
      entity:SelectPipe(0,"sniper_relocate_to", anchorName);
else
      AI:Signal(0,1,"NO_SPOT_FOUND",entity.id);
end
```


Entity script functions

Entity script functions are owned by the entities and must then have the syntax

```
<entity>:Function(params);
```

Where `<entity>` is an entity handler.
This means that, since these functions are owned by the entities, you don't need to pass the entity_id parameter as you usually do with AI system functions to affect an entity, but you must instead use the entity itself as the owner of the function (by using the ":" operator). For example, in a behaviour you would write

```
entity:ChangeAIParameter(AIPARAM_GROUPID, 100);
```

Inside a lua file of an entity (in Scripts/Default/Entities) you would instead write

```
self:ChangeAIParameter(AIPARAM_GROUPID, 100);
```

`entity:ChangeAIParameter``(parameter_index, value)`

Modify an AI parameter of an entity.

Parameters:

- (number) `parameter_index`: index specifying which AI parameter to modify. You may want to use one of the identifiers provided (see table below)
- (number) `value`: the value to assign to the AI parameter.

The following table shows which index identifiers can be used, which parameter they refer to and the values they can be assigned:

Parameter index	Entity's AI Parameter	Values
`AIPARAM_SIGHTRANGE`	Sight range (in meters). This affect how much the player should get close to be spotted by the enemy, and it's the minimum threshold in which the entity affect the player's SOM (perception level). See also Visibility determination details	0…9999
`AIPARAM_ATTACKRANGE`	Attack range (m). It's the maximum distance at which the enemy statrs to fire to the player	0…9999

AIPARAM_ACCURACY	Accuracy level; the lower it is, the higher is the AI's accuracy when firing at its target	0 … 100
AIPARAM_AGGRESION	Aggressiveness level	0…1
AIPARAM_GROUPID	Group id. This is used to identify groups of AIs in the level (see also Signal filters)	1..9999
AIPARAM_SOUNDRANGE	Hearing range (m). The maximum distance at which an AI can hear sounds	0…9999
AIPARAM_FOV	Horizontal field of view (angles). See section in Visibility determination details. To give the entity a 360° fov, set this parameter to a negative value.	-1..180
AIPARAM_COMMRANGE	Communication range (m). It's the maximum distance at which the entity can communicate with other AIs (see also Signal filters)	0…9999
AIPARAM_FWDSPEED	Forward speed for vehicles (m/sec)	0…9999
AIPARAM_RESPONSIVENESS	Responsiveness factor. The higher it is, the less time takes the AI to react to enemy presence	0…100
AIPARAM_SPECIES	Species id. An AI consider "enemy" whatever entity of another species.	1…9999

System script functions

System script functions are generic system functions owned by the common script object named System, which is the reference to CrySystem inside Lua.

```
System:Log(string_log)
```

Display a log line into the console

Parameters:
(string) `string_log`: the line to be displayed

See also:
`sprintf(..)`

Appendix C – Lua system callbacks reference

OnPlayerSeen(self, entity, fDistance)

Called when an enemy sees the player. When it happens, the entity has the player has its attention target.

Parameters:

- (table) self: current behaviour
- (table) entity: current entity
- (number) fDistance: distance at which the player has been spotted.

Example:

```
OnPlayerSeen = function( self, entity, fDistance )

   entity:SelectPipe(0,”approach_and_attack”);
   AI:Signal(SIGNALFILTER_GROUPONLY, 1, "PLAYER_SPOTTED",entity.id);

end,
```

OnEnemyMemory(self, entity)

Called when an AI stops seeing the enemy, but remember the position at which he saw it last. When this happens, the AI has this position as attention target (which is a dummy AI object).

Parameters:

- (table) self: current behaviour
- (table) entity: current entity

Example:

```
OnEnemyMemory = function( self, entity, fDistance )

   entity:SelectPipe(0,”investigate_target”);

end,
```

OnInterestingSoundHeard(self, entity)

Called when an AI hears an interesting sound (which is not threatening), for example the player jumping/walking etc

Parameters:
- (table) `self`: current behaviour
- (table) `entity`: current entity

Example:

```
OnInterestingSoundHeard = function( self, entity )
   entity:SelectPipe(0,"cover_look_closer");
   entity:TriggerEvent(AIEVENT_DROPBEACON); -- remember the sound
   position
end,
```

`OnThreateningSoundHeard``(self, entity, fDistance)`

Called when an AI hears a threatening sound, for example a gun shot or an explosion.

Parameters:
- (table) `self`: current behaviour
- (table) `entity`: current entity
- (number) `fDistance`: distance at which the sound has been heard.

Example:

```
OnThreateningSoundHeard = function( self, entity, fDistance )
   entity:MakeAlerted();
   entity:SelectPipe(0,"cover_investigate_threat");
   entity:InsertSubpipe(0,"cover_threatened");
end,
```

`OnNoTarget``(self, entity)`

Called when an AI stops having an attention target

Parameters:
- (table) `self`: current behaviour
- (table) `entity`: current entity

Example:

```
OnNoTarget = function( self, entity )
   entity:SelectPipe(0,"search_for_target");
end,
```

OnGrenadeSeen(self, entity, fDistance)

Called when an AI sees a grenade. The grenade becomes its attention target

Parameters:

- (table) self: current behaviour
- (table) entity: current entity
- (number) fDistance: distance at which the grenade has been spotted.

Example:

```
OnGrenadeSeen = function( self, entity, fDistance )

   entity:Readibility("GRENADE_SEEN",1);
   entity:InsertSubpipe(0,"cover_grenade_run_away");
end,
```

OnSomethingSeen(self, entity, fDistance)

Called when an AI sees something interesting, which is not the player nor threatening

Parameters:

- (table) self: current behaviour
- (table) entity: current entity

Example:

```
OnSomethingSeen = function( self, entity )

      entity:Readibility("IDLE_TO_INTERESTED");
      entity:SelectPipe(0,"cover_look_closer");
      entity:InsertSubpipe(0,"setup_stealth");
      entity:InsertSubpipe(0,"DRAW_GUN");
```



```
end,
```

OnJobExit(self, entity, sender)

Called when the current Job (in idle behavior) has finished

Parameters:

- (table) self: current behaviour
- (table) entity: current entity
- (table) sender: current entity

Example:

```
AIBehaviour.Idle_Talk = {
   Name = "Idle_Talk",
   NOPREVIOUS = 1,
   JOB = 2,

   CONVERSATION_FINISHED = function(self,entity,sender)
      entity.EventToCall = "OnJobContinue";
   end,

   OnJobExit = function(self,entity,sender)
      if (entity.CurrentConversation) then
            entity.CurrentConversation:Stop(entity);
            entity:StopDialog();
      end
   end
}
```

Appendix D – FAQ

This FAQ contains questions and answers used throughout the development of Far Cry. It was an evolving document and it's presented here in its entirety:

Art Related

How to make a bush hidable

How to make a bush move when a characters walks through it

How to make a bridge work with correct physics

How to make correct fences (flat 2-sided polys) with correct collision

Design Related

Enemies

How does the rear character work?
How does the cover character work?
How does the scout character work?
How do I make an enemy do something special when he sees the player?
How do I make an enemy use a mounted weapon?
How do I make an enemy throw a flare when he sees the player?
How do I make the enemy not notify his group immediately (automatically) when he sees the player?
How do I make the enemy sit down somewhere?
How do I make the enemy smoke a cigarette?
How can I make the enemy fix something?
How do I make enemies make a conversation?

Weapons

Where are the AI settings for the weapons read from?

General

What is an anchor and how can I place it?
How can I check whether I have triangulation/indoor nodes in my level?
When do I have to re-generate triangulation?
Do I have enough hiding places for a particular enemy?
Can I connect indoor waypoints from a VisArea to points in a navigation modifier?
How to set up the boids (like grasshoppers, frogs, glow bugs, etc.)

Troubleshoot:

Enemy behavior

Why does my offensive group attack NOT work?
I cannot get some enemies to STAY at a certain spot and not chase after the player...
Some enemies slide slowly without animating properly...

Art related

Why does my physicalized object do strange things?

How to make a bush hide-able?

You have to create additional planes in max (with the simplest possible geometry) that define where a player hidden in the bush is not visible. The rule of thumb is: **If the player has more than 40% of his screen covered with textures from the bush, he should not be seen** – because he believes he has hidden. You should experiment with the finished bush by placing it in the editor and checking whether it passes this one simple rule.

Most of the time – a fence of additional planes around the bush geometry will do the trick.

The additional planes must be of material **mat_obstruct**. The geometry must be **single sided** with the only side **facing the outside** of the bush, not the inside.

How does a bush move when a characters walks through it?

The object's "bending" parameter is multiplied with character's speed. Thus, if an object has a bending value of "0" it will not move. Make sure this value is an appropiate one, if you want a character to affect a vegetation.

How do I make an enemy do something special when he sees the player?

You have to place an anchor **within 30 meters to this enemy**, and select the **AIANCHOR_PUSH_ALARM** action for it. When the enemy sees the player, he will run to this point before he starts fighting with the player. You can place an AI only trigger here to trigger something special to happen.

If there is a group of people – **only one guy from the group** will go to the anchor (most likely the guy who sees the player first) – the rest will continue fighting normally.

Make sure that your anchor point is not within a forbidden area – in this case, the enemy will not be able to get to it.

This action can be combined with throwing a flare.

How do I make an enemy use a mounted weapon?

You have to place an anchor **within 2 meters from an actual mounted weapon**, and select the **USE_THIS_MOUNTED_WEAPON** action for it. Then any enemy **within 30 meters** of this anchor will prefer to use the mounted weapon when he sees the player and decides to engage him instead of normal combat.

If there is a group of enemies close to this weapon, only one will get to use it – the others will fight normally.

If the player advances to **less than 7 meters** to a guy using the mounted weapon, he will drop the mounted weapon and revert to his normal behavior.

How do I make an enemy throw a flare when he sees the player?

You have to place an anchor **within 10 meters** from the enemy, and select the **AIANCHOR_THROW_FLARE** action for it. The enemy will throw a flare when he sees the player and then revert to his normal behavior after that.

Make sure that the enemy you want to throw the flare **has FlareGrenade** ammo selected in his weapon pack. Otherwise he will throw whatever he has – at least a rock.

This action can be combined with running to alarm and delayed group alarming. The enemy will throw the flare and then run to the alarm.

How do I make the enemy NOT notify his group immediately when he sees the player?

You have to place an anchor **within 10 meters** from the enemy, and select the **AIANCHOR_NOTIFY_GROUP_DELAY** action for it. When he sees the player, the enemy will run to it and make an animation to notify his group of the player's position. If the enemy is killed before he reaches this anchor, his group will not be alerted.

This action can be combined with throwing a flare.

How do I make the enemy sit down somewhere?

You have to place an anchor **within 10 meters** from any enemy doing ANY job and select the **AIANCHOR_SEAT** action for it. The enemy will randomly choose this anchor sometimes and sit there for some time in between idle breaks.

Place this anchor **close to path points** of the enemy, or **close to the spawn spot** of the enemy. He only looks for this type of anchor when he stops at a path node (when he makes idle animations).

This anchor can be freely combined with other idle anchors. Any enemies close to them will choose them from time to time.

How do I make the enemy smoke a cigarette?

You have to place an anchor **within 10 meters** from any enemy doing ANY job and select the **AIANCHOR_SMOKE** action for it. The enemy will randomly choose this anchor sometimes and sit there for some time in between idle breaks.

Place this anchor **close to path points** of the enemy, or **close to the spawn spot** of the enemy. He only looks for this type of anchor when he stops at a path node (when he makes idle animations).

This anchor can be freely combined with other idle anchors. Any enemies close to them will choose them from time to time.

Where are the AI settings for the weapons read from?

You can find the file in the **Scripts\Default\Entities\Weapons** folder. The file is **AIWeapons**.lua.

How can I check whether I have triangulation/indoor nodes in my level?

You have to type **ai_debugdraw 1** in the console to enable the debug information of the AI system. Then you have to type **ai_drawplayernode 1** to enable drawing of the triangulation.

A triangle will appear around your position. As you move around the map, the triangle that you are currently in will be drawn.

Indoors, you will see the waypoint nodes and the links between them. If you don't see them, then generate AI triangulation. If this doesn't work, check whether you have an entrance to this indoor area.

When do I have to re-generate triangulation?

You don't have to regenerate triangulation often. These are the general guidelines:

- When you have added/removed/moved **OUTDOOR** brushes and/or vegetation.
- When you have changed the hidable flag of brushes and/or vegetation.
- When you have imported a layer that contains new brushes and/or indoor waypoints.

Re-generation of triangulation is not needed when you add/remove indoor waypoints. These are updated immediately (you can check this by enabling debug draw of triangulation/indoor nodes).

Do I have enough hiding places for a particular enemy?

You can check this by typing **ai_debugdraw 1** in the console. The enemy in question will have a **red question mark above his head** if he tried to hide but could not find a suitable hiding place. By adding hiding places in the area that he spawns it you can insure that this doesn't happen very often.

Why does my offensive group attack NOT work?

You need to have a **group of at least 3 people**. They all **must be in the same group** – please check their group ID's.

One of them has to have the Offensive leader archetype (or have the TLAttack2 character selected). The leader **has to have a job** – make sure he doesn't have some XXXIdle behavior (any job will do). Also make sure that all other group members have some sort of job – this is not a must, but it would look better.

If the attack kind-a works but not quite, make sure that you don't have some guy on the other side of the map that has the same group ID as your group – this will confuse the group (this will be fixed BTW).

Every member of the group can have ANY job behavior you want (he doesn't have to stand around). Even the leader can have any job behavior – please make an effort to make the enemies appear alive when they are idle.

Some enemies slide slowly without animating properly…

This **can only happen indoors**. Please check your waypoints. Some of them (around the sliding guy probably) are just a little below the floor, so that when I try to put them down on the floor, I end up going one level down from their current level.

The usual fix is to move the problematic waypoint up a little bit. In any case, if you cannot see more of the waypoint above the floor than below it, you should probably pick it up a little bit.

In general **you can place the points at any distance above the floor**; they will be projected on to the floor during path finding.

Can I connect indoor waypoints from a VisArea to points in a navigation modifier?

Yes, this is possible. All AI Points are the same and can be freely connected, no matter whether they are in a navigation modifier or in a VisArea.

Be careful however that **there is at least one entrance for every navigation modifier**, and one entrance somewhere in the indoor area. Not in each VisArea, but at least once in the same building.

You can quickly check whether your connections are working by following these simple steps.

How can I make the enemy fix something?

You can use the general **behavior Job_Fix**. You have to **place an anchor within 20 meters** from the enemy that you have assigned this job to, and **select the action AIANCHOR_FENCE** for it.

The enemy will approach, then start making animations like he is working on something approximately 1 meter from the ground. Occasionally he will stop and make some idle and then quickly get back to work.

What is an anchor and how can I place it?

An anchor is a **helper that designates a position** in space that is somehow special to the AI. It is used to help the AI when performing jobs (like fixing, smoking etc) and also to help the AI identify alarm spots, reinforcement spots etc.

You can **place an anchor using the AI button** in the editor. Click it and then select AI Anchor from the list. Now you can move over the map and click where you want to place this anchor.

Notice that the anchor has a distinct orientation – for some anchors, orientation is also important.

After you have placed it on the map, look in the properties for this anchor and you will find a field called Action. **Use the drop down dialog to select which action** to associate to this anchor.

You can also assign **anchor properties to a proximity trigger**. Look for the Action property in the properties of the proximity trigger.

Why does my physicalized object not behave correctly?

Make sure that this proxy has one smoothing group, and that the normals point in the right direction, outside, (otherwise the object may "shake" and behave very strangely, e.g. it wants to return to its initial pose and never comes to a stop).

This can only be solved by always applying RESETXFORM to the phys proxy before exporting.

The phys proxy should not be too small, otherwise the calculations will be wrong.
So make the proxy bigger than the actual object.

How to make a bridge work with correct physics

The physics calculation gets in kind of trouble, when it has to calculate very complicated objects colliding with vehicles. Detecting very "thin" collision objects the calculation for thin object is very time consuming therefore it is turned off by default. (otherwise we would have a waste of calc time, if we render thin and normal physobjects together)

The workaround for this problem: cut away the "road-part", make it a separate object with the same pivot as the base of the bridge. Give it a box (!!!) as collision. This works fine.

How to make correct fences (flat 2-sided polys) with correct collision

Do not switch on "2-sided" for the polygons that have the same texture on both sides (and have no separate phys proxy) nor use polys, containing open edges, as phys proxies. with "2-sided" switched on .

The physics system can not handle open edged 2-sided polys correctly.

Select these polygons, clone them (SHIFT + move) and flip the normals.

The benefits are a) correct physics and b) better lighting for backside/foreground.
Special Case: a swinging door should never have just two planes facing in opposite direction. Use a box instead. Otherwise, the door physics do not work correctly at all.

How to set up the boids bugs (like grasshoppers, frogs, glow bugs, etc.)

- You find the object in Entity/Boids/bugs
- With a shape connected to the boid they can be set up to a specific area. Without a shape the bugs may roam the whole island…
-
- **Most parameters are more or less self explanatory, but:**
- Behaviour is either 0: flying like flies or glow bugs, 1: flying like butterflies (with occasional stops), 2: jumps and movement on the ground (frogs, grasshopper)
- The "follow Player" checkbox doesn't mean they follow the player, but they are rendered only around the player.
- To adjust the "jump height" of frogs change the SpeedMin and SpeedMax to values like 0.5 and 4.
- Factor origin: the higher the value the closer the bugs stay to its starting objects. With a value of about 12 they can be bound to fly around a lamp (if the HeightMin and HeightMax frame the height of the lamp).

(added by Alex W.)

I cannot get some enemies to STAY at a certain spot and not chase after the player...

There are several ways to create a situation in which the enemies do not chase the player. In any case, there are special enemy personalities that do this, and special behavioral anchors that make this possible:

1. **Make the enemies have the REAR character**. The rear character does not move from his spawn position, and he can be made to protect a certain obstacle. Read more here. If you have a

problem with the rear throwing grenades, assign to him a weapon pack that contains no grenades.

2. **Place a defensive leader for the group**: If you place a defensive leader and make sure that *he perceives you faster than the rest* of the group, then he will make sure that no one leaves the protection site. He will do this even if he hasn't seen you but only heard you. By doing this you also provide an extra tactical choice to the player – killing the leader. You can learn how to place a defensive leader here.
3. **Close them in a forbidden area**: This is by far the most extreme case that ensures the enemies will not leave this position **in any case**. This should not be your first choice. Do this if you cannot get your desired behaviour with the other methods. *Understand that with this you are forever condemning the enemies to the confines of the forbidden area no matter of the player position.*

How does the REAR character work?

The Rear character works very differently from the cover or the scout. He is not so aggressive, he will not run after the player – he will rather stay where you have spawned him and maybe hide around that place.

Because of this, he is very convenient to have guard some passages, maintain barricades etc. He should not be placed in open places where more action is required because he doesn't tend to move much.

He throws grenades. He does this approximately 50% of the time, the rest he will shoot his normal weapon. If you do not include HandGrenade in his equipment pack, he will not throw any.

The rear will not immediately investigate disturbances… If the disturbance persists and changes nature (visual, sound etc), then he will investigate. He responds to a help request from a team member in the same manner that any other enemy does.

The rear character is not group-aware. Placing rears in offensive or defensive leader groups will not cause him to change his behavior.

You can place a tag point with the name <rearname>_PROTECT if you want to indicate a point that this particular enemy should protect. He will hang around that point.

If your REAR doesn't move at all after he has spawned, check if he has enough hiding points.

How does the cover character work?

The cover character is the main soldier type in the game Far Cry. He is versatile, converges quickly on the player. He should be used in points in the game where the player needs to be held back, and even motivated to retreat.

Covers use a lot of hiding objects. They never want to be caught open without any cover object between them and the player. This is why they need a lot of cover objects to work in a challenging way. You can check if your cover has enough hide points using this tool.

The cover is the most aggressive personality – he will always try to run to the player and kill him. He will use cover objects that take him closer to the player to advance, until there is no more cover object between the player and himself. Then he will stop and optionally lean or duck, but generally hold his ground. This happens until the player relocates into another position that gives the cover another object to advance to.

He does not flank, rather he chooses the fastest approach path to his enemy. He does not retreat, except when going for reinforcements or otherwise ordered (by a designer placing an anchor).

When the cover loses sight of the player in combat, he will chase him. He will use his last known position and run there looking for the player. They are hard to shake once they have noticed you.

You can modify the default cover behavior by placing different kinds of anchors. Consult this document for all the anchors that you can use to make some cover at a specific place do something special.

The cover is group aware and can be placed in a group with any leader. The covers are the "meat" of the group, so make sure your group always has mostly covers. Place a scout in the group where it would serve a purpose.

Covers can be placed alone and they will provide a challenge to the player. They will move a lot and pressure the player.

How does the scout character work?

The scout is one of the aggressive opponents – like the cover. The main differences between these two are in that the scout tends to keep a distance to the player. If the player comes to close, the scout **will turn his back and run to a safer distance**.

The scout also flanks all the time, choosing randomly left or right hiding objects from his target. They don't advance too much, they basically keep their distance.

Another difference is that the scout tends to investigate disturbances more than the cover, so place him where you want to simulate a point that is well guarded.

The scout is as group aware as the cover, although he performs a little differently in a group – but he will still listen to the team leader's orders.

Scouts placed alone will not provide sufficient resistance to the player. They are always "the support" section of each group. If they are alone the player can

chase them as they try to put distance between the player and themselves. This is why it is desirable that a group that contains a scout contains also "meat" – covers and rears that will keep the player pinned while the scouts seek out flanking positions.

Scouts will chase the player after they have lost him, but not so aggressively as the covers.

They should be lightly armed and they should generally have higher accuracy but less aggression. They should also always have longer sight ranges than that of the covers. That should compensate for their poor individual battle performance. As a rule of thumb, it should be more difficult to approach the scout then to fight him at short range.

How do I make enemies make a conversation?

There are 2 kinds of conversations AI can do, "random conversation" and "mission critical conversations". Random conversations are triggered randomly according to the applied jobs for the AI. They are not related to any special topic or mission. A random conversation could be a small chat about the weather or how awful mosquitoes are. Mission critical conversations are important for the player and are required to give the player information. They should be related to the mission or point in the story the player is now.

Random conversations:

Place an AIANCHOR_RANDOM_TALK. Place 2 or more AI's in the range of the anchor with properly chosen jobs and the same group ID. Adjust the anchor range by using the "Area" setting for the object. You will see at the drawn radius of the anchor if an AI will be in range or not.

The closest AI to the anchor will most likely start a random conversation (this is depending on the job, hence AI will just look for the anchor if they are idle).

The script for the conversation is located in:
MASTERCD\SCRIPTS\AI\Packs\Conversations\Idle.lua

```
-- FILLER A
	{
		Participants = 2,
		Script = {
			{
			Actor = 1,
			Duration = 1273,
			SoundData = { soundFile = "", Volume = 255, min = 10, max = 80,
		sound_unscalable = 0,},
			},
```

```
                {
                Actor = 2,
                Duration = 1362,
                SoundData = { soundFile = "", Volume = 255, min = 10, max = 80,
        sound_unscalable = 0, },
                },
        },
},
```

In the script you need to specify the amount of actors, the order the actors should act, the duration the part will take and sound file that has to be played. You can adjust the volume for each sound file there as well.

Mission Critical Conversation:

Place an AIANCHOR_MISSION_TALK. Place 2 or more AI's in the range of the anchor with properly chosen jobs and the same group ID. Adjust the anchor range by using the "Area" setting for the object. You will see at the drawn radius of the anchor if an AI will be in range or not.

The closest AI to the anchor will always start a conversation. The name of the anchor must be the same as the conversation name in the script. In the following example the name of the anchor must be: *training_pickup_B_1*

The script for the conversation is located in:
MASTERCD\SCRIPTS\AI\Packs\Conversations\Critical.lua

```
-- TRAINING
training_pickup_B_1 = {
        Participants = 2,
        Script = {
                {
                Actor = 1,
                Duration = 7113,
                SoundData = { soundFile = "", Volume = 255, min = 20, max = 100,
        sound_unscalable = 0, },
                AnimationToUse = "",
                },

                {
                Actor = 2,
                Duration = 1500,
                SoundData = { soundFile = "", Volume = 255, min = 20, max = 100,
sound_unscalable = 0, },
```

```
                AnimationToUse = "",
                },

                {
                Actor = 1,
                Duration = 630,
                SoundData = { soundFile = "", Volume = 255, min = 20, max = 100,
sound_unscalable = 0, },
                AnimationToUse = "",
                },

                },
        },
```

In the script you need to specify the amount of actors, the order the actors should act, the duration the part will take and soundfile that has to be played. You can adjust the volume for each soundfile as well. Additionally you can specify a special animation that should be used while the sound is played.

General Note: Again, the AI that is closest to the anchor will start the conversation and will be the first actor. To limit problems always make a small radius for the anchor and place it close to the AI you want to start the conversation. Notice that the AI will start looking for the anchor from the head (if you make the radius of the anchor 1 and place the anchor at the ground to the feet of the AI, the AI will not find the anchor, because the distance from the head to the position of the anchor will be more than 1 m, either place the anchor closer to the head or make the radius bigger).

If you want to have the AI make a special animation, make sure that the animation is done and the name listed in the *.cal file for the actor. Make sure all actors have all required animations. If you use more than 2 actors there is currently no way 2 specify the other actors. You can just make sure that the first AI is the AI you want to become the first actor.

