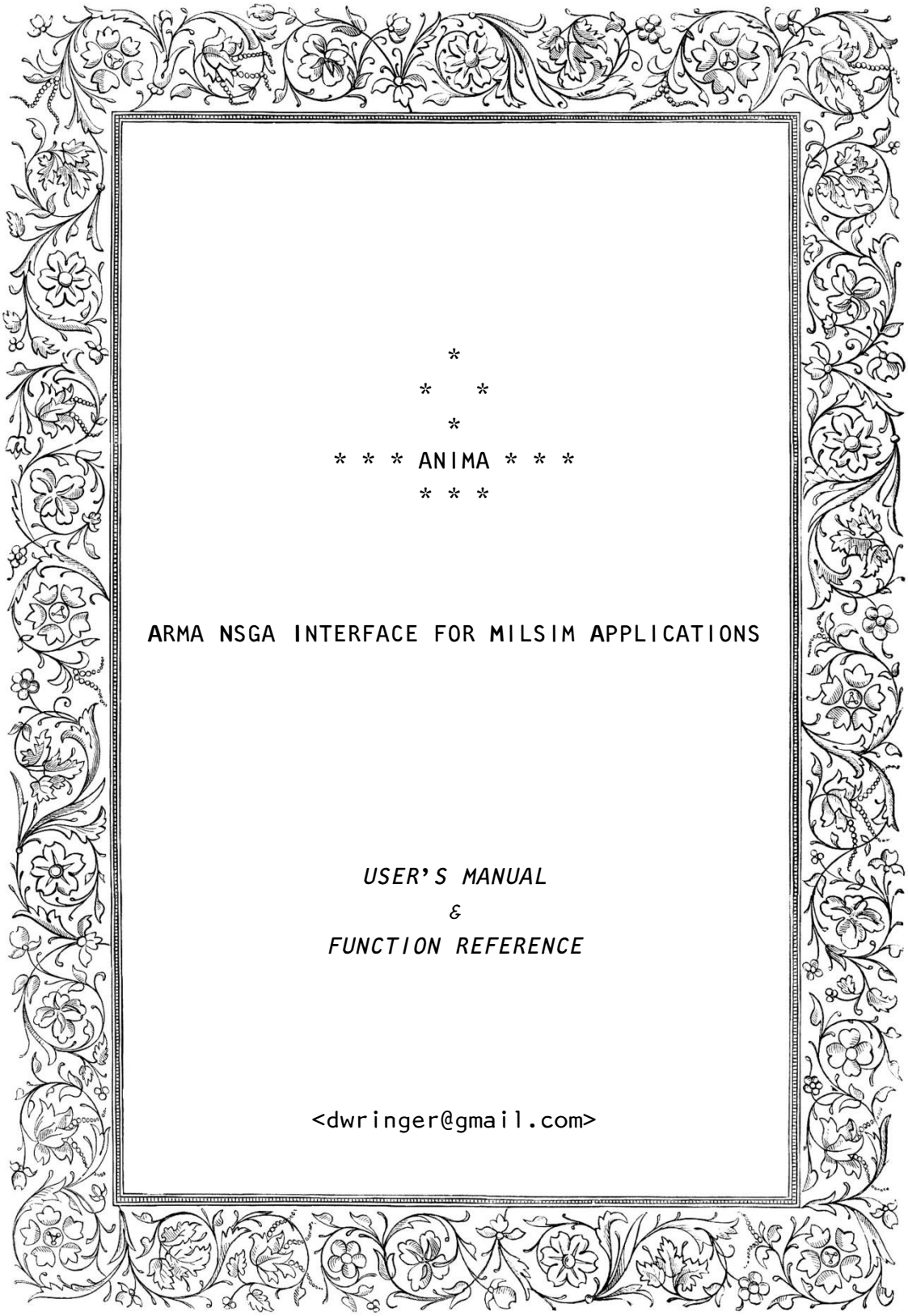
REL-220309-A



*
* *
*
* * * ANIMA * * *
* * *

ARMA NSGA INTERFACE FOR MILSIM APPLICATIONS

USER'S MANUAL
&
FUNCTION REFERENCE

<dwringer@gmail.com>

---FOR UNLIMITED DISTRIBUTION---

TABLE OF CONTENTS:

SETUP & BASICS

ANIMA is a library for mission designers and has endless applications, but it cannot be dropped in its present state into multiplayer missions. Experienced developers may be able to carefully adapt it by adding conditions in the right places. If anyone does this I would gladly welcome a pull request.

There are many features already built-in to ANIMA that can be used in single player missions with minimal effort. The most benefits are to be had, however, in advanced use of the position finding function, which requires some proficiency with SQF scripting.

I recommend starting with the entire set of scripts and then, later, removing those which are unneeded or unused. But, at the very LEAST, you will need the following folders and files in your mission folder:

```
\include\constants.h
\include\core.h
\include\optimizer.h
\classdef\ObjectRoot.hpp
\core\* <all contents>
\opti\* <all contents>
\init.sqf
```

Init.sqf must contain (in order!):

```
#include <include\core.h>
#include <classdef\ObjectRoot.hpp>
#include <include\optimizer.h>
ClassesInitialized = true;
```

This is sufficient to run the NSGA optimizer and fnc_find_positions, but it is highly recommended to use the full included init.sqf and all additional folders in order to see some fleshed out examples detailed in later chapters.

CHAPTER 1 -- INTRODUCTION

ANIMA is a collection of functions and scripts for providing AI groups with terrain/position analysis capabilities that can be used, for example, to dynamically create waypoints with tactical advantage or automatically land helicopters at suitably flat/clear locations.

This release is essentially an open-ended set of scripts for getting AI to do things in abstract terms like "land a helicopter nearby", find an overlook position with cover and concealment", "ambush nearby roads or intersections", "seek tactical advantage and engage targets"*, or "find a nearby place to unload troops".

These capabilities are implemented with a class system that exposes methods on initialized objects, called like so:

```
[<object>, "<method_name>"(, ...)] call fnc_tell;
```

The heart of ANIMA is the Optimizer class ("opti/Optimizer.hpp"), which is used to set up and control execution of the genetic algorithm. It's not necessary to use it directly, as its operation is encapsulated by fnc_find_positions (found in "opti/application_fns.h"). In turn, fnc_find_positions is very powerful but requires a lot of configuration and testing for different applications, so its functionality is encapsulated by the LocationFinder class, which is then leveraged by the higher-level Headquarters class ("classdef/Headquarters.hpp") for mission functionality.

*The order to seek tactical advantage and engage targets is particularly useful when given to several groups of units on each side, on triggers that repeat every couple of minutes and order the attack of all detected units of the other side. Thus multiple groups will continually try to outmaneuver one another and create immersive firefights that can last quite a long time.

CHAPTER 2 -- A FIRST EXAMPLE

As a quick example, we will demonstrate the "land_transports" method of the included "Headquarters" class. The "headquarters" is just a programmatically created Game Logic which stores state information about some units and provides some methods to deploy them.

In the editor, place a crewed transport helicopter Blu_heli_1, and an infantry squad whose leader's init contains the following:

```
{ _x assignAsCargo Blu_heli_1;
  _x moveInCargo   Blu_heli_1;
} forEach units group this;
```

Create a trigger with activation set to BLUFOR and an area covering the BLUFOR troops, and in its activation field put:

```
_thread = thisList spawn {
  waitUntil { not (isNil "ClassesInitialized") };
  Blu_HQ = ["Headquarters", _this] call fnc_new;
};
```

*Note that "thisList" above contains the BLUFOR troops, and gets passed (as "_this") to fnc_new as the first parameter used to initialize a new "Headquarters" object.

Finally, place a player unit somewhere else on the map, and provide a radio trigger with its activation set to:

```
_thread = spawn {
  [Blu_HQ, "land_transports",
   position player, [Blu_heli_1]] call fnc_tell;
  [Blu_HQ, "attack_targets", [player], 1] call fnc_tell;
};
```

Launch the mission and call the radio trigger. After 30-40 seconds the helicopter should start moving and then attempt to land in some

nearby field or clearing and let the troops out. The troops should then advance to a position that will ideally give them nearby cover, with intermediate distance and partially occluded line of sight to the player, and carry out a Seek & Destroy waypoint.

If you want to watch the positions on the map as they evolve in real-time, copy the file "classdef/Particle.hpp - MARKERS" over the file "classdef/Particle.hpp". This has an impact on performance and should only be used for testing and development. To disable the markers, copy back "classdef/Particle.hpp - HIDDEN" over "classdef/Particle.hpp".

CHAPTER 3 -- HOW DOES IT WORK?

Let's say we want to find a position to place an invisible helipad near a given location, to give a transport unload waypoint to an AI helicopter. We would do this manually by finding a position that:

1) is nearby
2) is flat
3) doesn't have objects/clutter in the area
4) doesn't have easy line-of-sight to hostile areas

ANIMA can do this automatically. It takes a look at some random positions in an area and scores them in all 4 areas: distance, flatness (how much does the Z-coordinate vary in the area surrounding each point?), object clutter (how many objects are within a certain distance?), and line-of-sight (how visible is each point from a predefined set of "enemy" points?). Over a series of generations, the worst positions are discarded and new ones are added relative to the existing ones, and ultimately we are left with a set of points that scored relatively well in all areas. Thus we can programmatically find potential landing zones anywhere on the map at any time during a mission.

The same technique can be used with different objective functions to find positions for other criteria, with the only real limits being the ability of the user to objectively specify what makes a "good" position, and the performance of the algorithm itself, which can be quite slow and requires each application to be optimized as much as possible. In particular it is important to find the simplest and most effective objective functions, and keep the population size and generation count as low as possible while obtaining usable results.

CHAPTER 4 -- FURTHER DETAIL

An NSGA is a "non-dominated sorting genetic algorithm": in essence, this is a technique for evaluating a series of vectors (in this case, positions in the Arma world) across multiple objectives (functions that assign various scores to the positions based on defined criteria). Those positions which have the best scores in any one or more objectives are "cross-bred" to find new, potentially improved candidate positions while the worst positions are dropped. This is repeated for multiple generations until suitably high-scoring positions (across most or all objectives) have been found.

In ANIMA, the initial population of candidate positions is initialized using one of a few different "shape functions" (ring, concentric rings, cross) which return a formation of points around a specified point. This is most often used around a unit or location to begin finding positions nearby. "Cross-bred" positions offset from existing positions based on the distances between other existing positions, meaning that they can get closer or farther away depending on how much the other points are clustered together (ANIMA does this incorporating a method known as "differential evolution").

If the first generation of positions scores best in a small area of the map, subsequent generations will tend to search around that area; however, if the highest scoring positions are spread out then subsequent generations will continue to spread, leap-frogging to new areas in search of better scores. When most of the farther areas are discarded as inferior then the algorithm will start to converge. It's typical that no single best position is ever found, and it's most practical to cut off after a few generations and take the set of points that scored highest in as many objectives as possible. It can be beneficial to include an objective function that scores highest at a certain distance from a unit or location, thus promoting the algorithm to converge at that distance.

CHAPTER 5 -- USING CLASSES

ANIMA makes extensive use of a custom class system for object-oriented method dispatch on game objects. As a hypothetical example, let's assume the existence of a "Dictionary" class with methods to set values, get values, list all stored keys, and retrieve all stored [key, value] pairs. We can initialize a Game Logic in the editor as a Dictionary instance like so:

```
_nil = this spawn {
  waitUntil {not isNil "ClassesInitialized"};  // Required at start!
  [_this, "Dictionary"] call fnc_instance;
};
```

Classes can also be instantiated without a preexisting editor object, using fnc_new. Each new instance will be created as a Game Logic in a dedicated group called Group_ClassLogic. This is done like so:

```
// General form:
_newObject = ["ObjectClassname"<, parameters, ...>] call fnc_new;

// Dictionary class example (constructor doesn't take params):
MyDict = ["Dictionary"] call fnc_new;
```

Methods are called by sending the method name as the first parameter after the object to fnc_tell:

```
// Add a key, value to the dictionary:
[MyDict, "set", "test key", "some test data"] call fnc_tell;

// Look up the stored value from a key:
_v = [MyDict, "get", "test key"] call fnc_tell;

// Get all keys stored in the dictionary:
_keys = [MyDict, "keys"] call fnc_tell;

// Get an alist of all key, value pairs stored using this interface:
_kvps = [MyDict, "items"] call fnc_tell;
```

Note that we use fnc_tell with Arma's call, which blocks execution. Never use call directly in a trigger or waypoint without wrapping it inside of a new spawned context. This is employed like so:

```
_thread = [] spawn {
  [<obj>, "<method>", ...] call fnc_tell;
};
```

In include\classes.h there is a series of macros to facilitate the construction of class definition files, as found in classdef\ in Dictionary.hpp and many others. These macros are not required, but provide a slightly cleaner syntax for making a file with several class or method definitions. Without using the macros, it is possible to declare and instantiate classes on-the-fly, while the simulation is running.

CHAPTER 6 -- INCLUDED MODULES

While users are encouraged to envision and implement their own uses of fnc_find_positions and the Optimizer class, it can be daunting to begin experimenting and non-trivial to create working applications. For this reason I have included some modules (implemented with classes as described in Chapter 6) that cover some common use cases in single player scenarios.

The HEADQUARTERS Class

The Headquarters can technically be used to control groups of an entire side, but it's mainly intended to manage a subset of groups which are expected to work as a cohesive unit.

Readers should be familiar with this example from Chapter 2 - create a trigger with activation set to BLUFOR and an area covering some BLUFOR troops, and in its activation field put:

```
_thread = thisList spawn {
  waitUntil { not (isNil "ClassesInitialized") };
  Blu_HQ = ["Headquarters", _this] call fnc_new;
};
```

A Headquarters keeps track of infantry groups, vehicles, and loaded transport vehicles. Methods that demonstrate ANIMA functionality include:

approach_targets

```
[Blu_HQ, "approach_targets",
 <target-list>, <num-transports-to-send>,
 (<maintain-awareness?>), (<nearest-frequency>)]] call fnc_tell;
```

Loaded transport vehicles approach a target and unload somewhere out of sight at medium range, intended to maximize nearby cover and occluded lines of sight to the targets. The last two parameters are

optional. Maintain-awareness has no effect, vehicles will always travel at full speed in careless stance. Nearest-frequency takes a value from Ø-1 that gives the percentage chance that only the nearest found waypoint locations will be used. When less than 1, transports are more likely to drive past their target to find a spot to unload.

attack_targets

```
[Blu_HQ, "attack_targets",
 <target-list>, <num-groups-to-send>] call fnc_tell;
```

Groups (infantry as well as vehicles) approach the target(s) and perform a Seek & Destroy waypoint at a location at relatively close range, chosen to have cover and occluded lines-of-sight to target(s).

insurgency_arm

```
[Blu_HQ, "insurgency_arm",
 <ambush-target>,
 <ambush-search-radius>,
 <ambush-type>,
 <activation-probability>,
 (<enemy-observers>),
 (<engage-now?>),
 (<insurgent-ratio>)] call fnc_tell;
```

This functions alongside the mkcivs module (described later). Finds spots within ambush-search-radius from which to engage a given ambush-target (unit or Game Logic). A percentage of civilians (insurgent-ratio) will go to these positions, join OPFOR, equip weapons and perform Seek & Destroy when the ambush is engaged. Returns an object representing the armed ambush, which can be triggered with the "insurgency_engage" method. The ambush-type is either "intersections", "roads", or "units". "Intersections" and "roads" optimize cover relative to road/intersection positions, while "units" uses the target directly. Enemy-observers is an optional list of units to avoid when finding ambush locations.

land_transports

```
[Blu_HQ, "land_transports",
 <position>, <helicopters>,
 (<enemies>), (<radius>), (<n-steps>),
 (<search-population-size>), (<init-shape>)] call fnc_tell;
```

Sends loaded transport helicopters to land near position, optionally avoiding enemies, where they unload and then wait. Initializes search with given radius, maintains search-population-size candidates during evolution and starts them in pattern given by init-shape. The last five arguments are completely optional and typically not required except in special circumstances.

land_helicopters

```
[Blu_HQ, "land_helicopters",
 <position>, <helicopters>,
 (<enemies>), (<radius>), (<n-steps>),
 (<search-population-size>), (<init-shape>)] call fnc_tell;
```

This method works exactly like land_transports, but doesn't unload any passengers (uses MOVE instead of TR UNLOAD).

Other potentially useful Headquarters methods (non-ANIMA) include:

leaders

```
[Blu_HQ, "leaders", (<named-only?>)] call fnc_tell;
```

Returns an array of all living group leaders tracked by the Headquarters. If named-only is true, then only named units are included.

safety_protocol

```
[Blu_HQ, "safety_protocol", <helicopter>] call fnc_tell;
```

This method simulates an emergency parachute for the player, to be used when riding in a helicopter. If the player ends up outside the vehicle above a certain height AGL (often a result of critical damage), then a parachute will be spawned for the player to have some chance of survival.

watch_for_smoke

```
[Blu_HQ, "watch_for_smoke", <helicopters>, <whose>] call fnc_tell;
```

Monitors <whose> for the throwing of a smoke grenade, which is then watched until stationary. Once it stops moving, the grenade position is passed to the land_helicopters method so they will land nearby.

orbit_position

```
[Blu_HQ, "orbit_position",
 <position>, <helicopters>, <radius>, <altitude>,
 (<bearing>), (<counter-clockwise?>),
 (<n-points>), (<star-pattern?>)] call fc_tell;
```

Begins a loop in which helicopters are assigned a circular pattern of waypoints on a continually repeating basis, thus orbiting a given position at a given radius/altitude. Several parameters are optional: Bearing represents initial offset, and star-pattern, when true, causes the helicopter to fly to the next waypoint on the opposite side of the circle each time (works well with helicopters that have forward-facing cannons).

orbit_group

```
[Blu_HQ, "orbit_group",
 <group,-units,-or-position>, <helicopter>, <radius>, <altitude>,
 (<counter-clockwise?>), (<n-points>), (<star-pattern?>),
 (<recenter-distance>)] call fnc_tell;
```

This works like orbit_position, but tracks a group instead. Orbit is updated once the group's average position moves more than recenter-distance. Works well after a TR UNLOAD to provide air support.

holding_arc

```
[Blu_HQ, "holding_arc",
 <focus-group>, <helicopter>,
 <angle-cw-bound>, <angle-ccw-bound>,
 <radius-inner>, <radius-outer>, <altidude>,
 <counter-clockwise?>, <n-points>] call fnc_tell;
```

Employs orbit_group in two directions, to cause helicopter to fly back and forth between bearing angle-cw-bound and angle-ccw-bound, at radius-inner on the inside and radius-outer on the outside. This brings the side guns to bear on a target from a specific direction and can serve after a TR UNLOAD as more effective air support than orbit_group. If the player is in one of the guns then using a large outer-radius can keep the helicopter out of danger when the helicopter turns around and the player faces away.

abort_pattern

```
[Blu_HQ, "abort_pattern", <helicopter>] call fnc_tell;
```

Sets a value on the helicopter to break it out of orbit/holding arc. Please wait after using this for the helicopter to return to idle (use getVariable on the helicopter to ensure both "orbitIdle" and "arcIdle" are either true or undefined - not false - in the helicopter's namespace) before invoking another orbit or holding arc method.

The MKCIVS Module

This module was created back in the days of Arma 2 as an alternative to BIS's ambient civilians module. Civilian zones are placed as Game Logics in the editor, and initialized as instances of the CivilianZone class.

Each civilian zone creates two triggers - one to spawn civilians when the player (or NAMED unit) gets within a certain range, and one to despawn them again when the tracked units go out of a certain range (the ranges can be set independently). Civilians randomly populate the demarcated area and wander around with a DISMISSED waypoint. When they wander outside a predefined range, their groups are reinitialized, causing them to walk back to where they started and start the DISMISS waypoint all over again. Thus civilians wander around endlessly within a small area.

This module is tightly integrated with the arm_insurgency and engage_insurgency Headquarters methods. (These behaviors are also available without a Headquarters by using the KillZone class). Each civilian has a unique name based on its zone, and civilians can be assigned a callback function for their "Killed" event handlers.

The CIVILIANZONE Class

A CivilianZone is initialized by taking a Game Logic from the editor, giving it a unique name (like civ_zone_1), and instantiating it with the following init code:

```
_thread = this spawn {
  waitUntil { not (isNil "ClassesInitialized") };
  [_this, "CivilianZone",
   <n-groups>, <max-group-size>, <civ-radius>,
   <spawn-radius>, <despawn-radius>] call fnc_instance;
};
```

CHAPTER 7 -- THE LOCATIONFINDER CLASS

The LocationFinder class sets up and makes calls to fnc_find_positions to cover a variety of common use cases. Some of them work better than others, but this class should serve as an example of how one might create new applications, or a starting point for further development.

LocationFinders do not need to be instantiated from existing objects, it's enough to use fnc_new. Use the constructor like so:

```
MyLocFinder = ["LocationFinder"] call fnc_new;
```

The constructor can take one or two additional parameters. The first specifies population size (the default is a very conservative 5, which is fast but not particularly effective - larger populations are slower but much more meticulous in finding good positions). The second is a "shape function".

A shape function is a function of a number *n* for creating a list of *n* points ([x, y, z]) in a particular 2D shape (normalized so each component is between -1.0 and 1.0, inclusive). The default shape function, fnc_make_ring, is just a circle around the center. Other choices are fnc_make_square, fnc_make_concentric_rings, fnc_make_cross, or fnc_make_ring_cross (creates five rings in a cross pattern). These shapes form the initial distribution of search positions.

For example, to create a LocationFinder that uses a population size of 15, in five groups of 3 arranged in a "+" pattern, use:

```
MyLocFinder = ["LocationFinder", 15, fnc_ring_cross] call fnc_new;
```

LocationFinders may also be configured with their "configure" method:

```
[MyLocFinder, "configure", <list-of-key-value-pairs>] call fnc_tell;
```

Configurable keys of the LocationFinder include:

```
  populationSize : size of the search population
  shape : this is initialized with [<pop-size>] call <shape-function>;
  radius : How much do we enlarge the shape? (Default: 105)
  objectives : List of Particle objective functions
  assignments : List of key-value pairs to set on each Particle
  center : Position at which to center the Location Finder
  color : Color to use if visible markers are enabled
```

It isn't really necessary to use "configure" - if users desire such fine-grained control then I would recommend using fnc_find_positions directly.

The LocationFinder instance must be centered on an initial location prior to operation. This is done with the "set_location" method.

set_location

```
  [MyLocFinder, "set_location", <position>] call fnc_tell;
```

Next, LocationFinder instances must be configured with objective functions that define the optimal characteristics of positions to be found. Methods to automatically set these functions for specific goals include:

obj_level_terrain

```
  [MyLocFinder, "obj_level_terrain"] call fnc_tell;
```

This method optimizes for positions that have the flattest surrounding area, determined by testing a 5x5 grid of points covering a 12x12m square.

obj_landing_zones

```
[MyLocFinder, "obj_landing_zones"] call fnc_tell;
```

This method optimizes for level terrain that is not water, is ideally not closer than 40m to buildings/trees/large objects, and has unobstructed lines-of-sight within a few meters of the position (using a grid method like the one used to check level terrain).

obj_tactical_approach

```
[MyLocFinder, "obj_tactical_approach", <target-list>] call fnc_tell;
```

This method sets up to assign each optimizer particle with a list of targets, and optimizes for non-water positions that are not in enemy LOS, are ideally about 600m from the targets, have surrounding objects for cover and concealment, and ideally have roads nearby.

obj_tactical_advantage

```
[MyLocFinder, "obj_tactical_advantage",
 <target-list>, (<avoid-water?>] call fnc_tell;
```

This method sets up optimization of positions near buildings, large objects, and dense vegetation, about 50 to 75m from targets, with full or partial line of sight to the targets.

obj_sniper_nests

```
[MyLocFinder, "obj_sniper_nests",
 <target-list>, (<avoid-water?>)] call fnc_tell;
```

This works like obj_tactical_advantage, but ignores buildings and for distance simply optimizes for being at least 300m away.

RUNNING the LocationFinder

In order to get positions from the LocationFinder after initializing and setting it up, we need only call the "run" method with one or two parameters.

```
[MyLocFinder, "run", <n-positions-required>, (<step-target>)]
 call fnc_tell;
```

The first parameter is how many positions you need. Sometimes the algorithm might only find two or three. If you need more, the algorithm will run again until it has enough. Omit this parameter to get all the positions found in one run.

The second parameter is the target number of steps for the genetic algorithm. This is configurable so if the positions being found are inadequate, increase it. If the positions are fine but taking way too long, decrease it.

CHAPTER 8 -- OBJECTIVE FUNCTIONS

In order for the ANIMA engine to evaluate positions in the world for fitness, it must be supplied with a list of objective functions. Each of these is a function that takes a Particle (Game Logic with a position in 3D space) as its only parameter, and returns a value between 0 and 1 (perhaps counterintuitively, 0 is considered the most fit, and 1 the least). Any conceivable function that meets this definition can be used.

One common technique is to use setVariable on the Particles to assign each one with a list of targets (fnc_find_positions takes a list of ["<variable>", <assignment>] pairs as one of its parameters and calls setVariable on each Particle this way). Then, in the objective functions, use getVariable to retrieve the list and perform, for example, LOS or distance checks to the targets.

Included with ANIMA is a function called fnc_normalizer, which takes three arguments:

```
[<particle-function>, <min-expected-value>, <max-expected-value>]
```

This is so we can write particle functions that return a range outside of 0-1. The normalizer function returns a version of the original function that does constrain its results to the 0-1 range.

Example: Say we have an objective function that counts the number of targets with LOS to the Particle. It's easiest to just write it to return the count directly, so with 5 targets it would return a value between 0 and 5. Let's call it fnc_count_los. To get a function limited to the range we want, we invoke:

```
_fn = [fnc_count_los, 0, 5] call fnc_normalizer;
```

Now _fn is a function of one particle that returns a value between 0 and 1 and is suitable for use as an objective function in ANIMA.

There is a problem with our normalized function, though: ANIMA seeks to *minimize* the value of its objective functions. Thus having 5 or more LOS to target in our fnc_count_los example would return 1.Ø, and be considered *least fit*.

We could write our function to invert this, but that's not necessary. Another included function incorporates fnc_normalizer, but is more powerful: fnc_to_cost_function. This is called like so:

```
_fn = [<maximize?>, <min-expected-value>, <max-expected-value>,
       '[_x]', <original-fn>] call fnc_to_cost_function;
```

The first parameter is a boolean: if set to true, then the returned function is inverted for us. Thus:

```
_fn = [true, Ø, 5, '[_x]', fnc_count_los] call fnc_to_cost_function;
```

will give us a function that returns Ø when there are 5 or more lines-of-sight, and returns 1.Ø when there are Ø.

The unsightly '[_x]' string is the parameters passed to fnc_count_los, in string form, where _x represents the particle being tested. This allows us to use fnc_to_cost_function and change a function that expects a Particle plus additional parameters into a function that only takes a Particle and has the other parameters "baked-in", so to speak.

```
OPT_fnc_distance_from_player =
        [true, 5Ø, 3ØØ,
        '[position _x, position player]',
        fnc_euclidean_distance] call fnc_to_cost_function;
```

This example from opti/objective_fns.h changes fnc_euclidean_distance, a function that takes two positions as arguments, into a new function that takes only a single Particle as a parameter. The new function calls fnc_euclidean_distance with the Particle (_x)'s position as the first parameter and the player's position as the second parameter, and

returns a number from Ø to 1: Ø.Ø if the distance is 3ØØ or greater, and 1.Ø if the distance is 5Ø or nearer.

With this technique, we can also create basic evaluation functions that take a Particle and a radius, and then turn them into multiple objective functions that operate at different radii.

```
CLASSES

Optimizer(pop_size, particle_classname, [particle_color])
.get_position()
.position_offsets()
.set_position()
.uniform_position(position)
.report_position()
.conform_positions(positions)
.conform_units(units)
.fit_terrain()
.do_nothing()
.perturb(n, radius)
.radial_scatter_2d(r_min, r_max)
.displace_shape(points, heading, scale)
.ring_out(radius)
.rings_out(radius, leaves)
.add_objective(objective_fn)
.evaluate_objectives()
.de_candidates(weight, frequency)
.non_dominated_sort()
.sorted_average_distances_3d(subpopulation)
.sorted_average_distances(subpopulation)
.moea_step(candidate_generation_method, candidate_generation_params,
           preevaluation_method, preevaluation_params,
         bin_creation_method, bin_creation_params,
         bin_ordering_method, bin_ordering_params)
.MODE_step(weight, frequency)

Marker(position, shape, type, size)
.set_alpha(alpha)
.set_brush(brush)
.set_color(color)
.set_direction(direction)
.set_position(position)
.set_shape(shape)
.set_size(size)
.set_text(text)
.set_type(type)
```

```
.redraw()
.show()
.hide()

Particle(color)
.set_position(position)
.get_position(position)
.add_objective(objective_fn)
.evaluate_objectives()
.differential_evolve(other_particle, adjunct_particle,
moderator_particle, weight, frequency)

LocationFinder()
.set_location(location)
.obj_level_terrain()
.obj_landing_zones([avoid_list])
.obj_tactical_approach(targets)
.obj_tactical_advantage(targets, [avoid_water?])
.obj_sniper_nests(targets, [avoid_water?])
.run(max_required_positions, target_step_count)

Headquarters([units])
.track(units)
.untrack(units)
.dispatch(n, locations, full_speed?, no_transports?)
.dispatch_transports(n, locations, full_speed?,
nearest_only_frequency)
.radio(msg)
.available()
.available_groups()
.available_transports()
.available_pairs()
.moveable_vehicles([refresh_lists?])
.living_men([refresh_lists}])
.men_from_trigger(trigger_list)
.purge()
.retrack()
.leaders([named_only?])
.regroupify(groups, waypoints)
.approach_targets(targets, n, [limited_speed?, [nearest_frequency]])
```

```
.attack_targets(targets, n, [limited_speed?])
.safety_protocol(helicopter)
.insurgency_arm(obj, radius, type, chance, watchlist, [engage?,
[ratio]])
.insurgency_engage(armed_zone, pull_radius, percentage)
.woodland_vests(men)
.land_transports(position, helicopters, [enemies, [search_radius,
[search_steps,
                 [search_population, [shape_fn]]]]])
.land_helicopters(position, helicopters, [enemies, [search_radius,
[search_steps,
                 [search_population, [shape_fn]]]]])
.watch_for_smoke(helicopters, whose)
.orbit_position(position, helicopters, radius, altitude, bearing,
ccw?, n, star?)
.holding_arc(focus_group, helicopter, angle_cw_bound, angle_ccw_bound,
             radius_inner, radius_outer, altidude, ccw?, n,
called_recursively?)
.orbit_group(focus_units_or_pos, helicopter, radius, altitude, ccw?,
             n, star?, recenter_distance)
.abort_pattern(helicopter, called_recursively?)

UnitGroup()
.center_pos()
.add(unit)
.remove(unit)
.groups()
.reinitialize_groups()
.move()
.sequester()
.desequester()
.timed_patrol()
.add_waypoint()

CrewUnitGroup()
.assign(unit, role)
.auto_assign(units)
.init_in_place(men)
.board()
.board_instant()
```

```
.is_serviceable()

CivilianZone(group_count, group_size, zone_radius, spawn_radius,
despawn_radius)

KillZone(type, radius, probability, targets, side)
.arm(population_size, generations)
.engage(pull_radius, probability)
```