Proposal for short term goals for DCS Mission Scripting Environment enhancements

Introduction

This document has been created as a result of a discussion on the Hoggit subreddit with Eagle Dynamics' NineLine and aims to fulfill the request of:

> "Do up a document with say your top APIs you would like to see, what you need out of them and possible uses for it. How it would benefit the community, etc. Seriously, be over descriptive, the more the better. Build me a list and I will discuss with Kate. I can't promise anything, but if you give me greater details on what you think is needed, I will see what I can do." - *NineLine*

We hope that this document provides enough information to Eagle Dynamics to provide a solid foundation for prioritization of the proposals specified in this document in the short-term. We also hope that this document prompts Eagle Dynamics to consider the wider array of requested APIs, forum API requests, and future requests for prioritization as a longer-term goal.

This document has been reviewed by around 30 community members and endorsed by admins of the following online servers:

- Hoggit
- Enigma's Cold War Server
- Growling Sidewinder
- Flashpoint Levant
- Conquest DCS
- Task Group Warrior
- 473rd Squadron
- MMSERVERSMACK
- [KW]LaTaniere
- SK Dedicated Server
- DCS Singapore

As well as the following DCS related projects:

- LotATC
- DCS:Liberation
- Skynet IADS
- DCS-gRPC
- OverlordBot

Glossary

MSE: Mission Scripting Environment, also known as the SSE (Simulator Scripting Engine). The lua scripting environment that is accessible from lua scripts run as part of a `Miz` file, whether embedded in the `Miz` or loaded from the filesystem as part of the `Miz` startup.

Modding: Modifications made to the structure of DCS by unofficial third parties to enable new functionality. Examples of these are new unofficial aircraft, new ground models, additional exports to cockpits, etc. *It is important to note that the authors of this document do* ***not consider anything to do with the MSE to fall under the Modding heading*** *as unofficial third parties are not modifying the structure of DCS but requesting that Eagle Dynamics officially make changes.*

Background of MSE development

Eagle Dynamics created the Mission Scripting Engine when DCS World was around version 1.1.0. During this process, around 1.2.0, an Eagle Dynamics developer directly engaged with Grimes, Speed, and a few other DCS community members on what APIs would be desired. This developer also wrote documentation to the official Eagle Dynamics wiki and documented the changelog page for the MSE.

This work stopped when DCS 1.2.6 was released, which is around the time that this employee left the company. This indicates, along with the state of the MSE, that the work was left in an unfinished "good enough" state rather than the project reaching a set milestone whereby a considered decision was made that the work was deemed "completed" and work on the MSE, especially community requested ones, has almost stalled for a decade.

Looking at the added functions in the changelog, some of which were suggestions by the testers, we see a number of the most important ones to scripters such as coalition.addGroup. If a random 10% of what was added between 1.2.4 and 1.2.6 were removed almost every mission that uses the MSE would fail. Scripting is the reason it can be argued that DCS has dynamic campaigns. All of the top servers use the MSE to one degree or another. Sometimes it's just for the server API and moderation, sometimes it's more involved with dynamic, persistent campaigns or simply accomplishing something easier via script than can be done in the editor. Even some DLC campaigns use scripting.

A recent example of the windfall Eagle Dynamics gains from API creation is the launch of the Enigma Cold War Server (ECW). In late 2021, Eagle Dynamics released scripting APIs that allow for shapes and text visual objects to be drawn on the F10 map. The ECW makes use of these APIs to display campaign information to players in the form of frontline "sectors" and who currently owns them based on color information. Enigma, the server owner, confirms that without this ability to draw and update these objects mid-mission, then the server would never have been created. Since February 14th, 2020 over 2,500 unique players have played on the server.

Proposed Areas of Improvement

Rather than provide an unsorted list of the outstanding requested APIs and events, we will group a subset of them by functionality where we hope that Eagle Dynamics can achieve the greatest benefit for the minimum amount of work as a short term goal. Implementing these proposed areas will allow for new game-loops and more interactive missions for players.

Each of the following proposed areas of improvement can be evaluated in isolation from each other and, if completed, would provide benefit to players on their own. The authors do not expect all of the below proposed areas to be completed in for a single release but rather hope for several releases over time.

1. Map information

The maps of DCS are largely opaque to scripts in the MSE which means that mission makers have to manually determine safe areas for creating objects and units. This is to avoid causing mission breaking issues such as objects and units unintentionally appearing inside a forest or being clipped onto buildings.

Having these manually selected "safe" areas requires a lot of manual work which reduces the number of locations that objects can be spawned at, as mission makers have finite time and energy to spend on performing this task. They also have to be reviewed whenever the maps are updated.

These APIs below are intended to allow coders to safely spawn objects without needing to pre-select spawn areas or manually update them when maps change. All of these APIs are read-only and do not affect gamestate so most of them are the simplest to implement in this document.

An example of where this would prove useful would be servers such as Lima Kilo's `Flashpoint Levant` where missions are requested on-demand by players. Currently, the units in these missions have to be manually positioned by the mission scripters, which leads to players eventually memorizing the mission locations. This greatly reduces replayability and freshness for longtime players.

However, with a map whose information is accessible from code, it would be possible to do things such as:

1. Get a random town that is within a specified distance of a player
2. Get a random point a specified distance from this town on a random bearing
3. Get the nearest road to this random point
4. Spawn a convoy on that road with a route that takes it into the town
5. Provide the player with a coherent briefing using `OutTextToCoalition` (or group) based on this dynamically spawned convoy with position information

It will also allow players to more easily create zones that conform to the shape of a town, which can then be used elsewhere in the scripting environment, for example determining ownership of the town based on units inside the town shape.

Get All Towns

This API returns a list of all essential details of all the towns on the map. Due to the potential size of the response, this API would seldom be called, but would allow exporting of map information for use in external clients such as `LotATC`, mission generators such as `DCS:Liberation` and `RotorOps`, `DCS-gRPC` clients, online maps etc.

Method signature

`land.getAllTowns()`

Return value We assume that towns have IDs. This may not be the case, in which case the `id` field is not required. Some of this data exists already within the towns.lua file for each map and is used with the KA-50 ABRIS avionics and simply features the coordinates in lat/long and a localized display name.

```
{
    [1] =
        {
            ["id"] = int, -- Integer ID of the town (Might be a string that
                          -- is just the normalized town name)
            ["name"] = string, -- Human Readable name of the town
            ["center"] = vec2, -- Calculated center of the shape of the town
            ["shape"] = { -- List of points that form the shape of the town
                [1] = vec2,
                [2] = vec2,
                [3] = vec2,
                ...
            }
        }
    [2] = {
    ...
    }
}
```

Get Town by ID

This API returns the essential details of a specified town. This assumes that towns have IDs associated with them. It is possible that they do not and this API will not be implementable and instead we use `land.getTownByName` below

Method signature

```
land.getTownById(
    integer -- DCS ID of the town
)
```

Return value Returns the same result as `land.getAllTowns()`but only for the specified town. This might be redundant if the ID specified above is actually the name of the town anyway.

Get Town by Name

This API returns the essential details of a specified town.

Method signature

```
land.getTownByName(
    string -- Name of the town
)
```

Return value Returns the same result as `land.getAllTowns()`but only for the specified town.

Get Nearest Town

This API returns the essential details of the nearest town to a given vec3 point.

Method signature

```
land.getNearestTown(
    vec2
)
```

Return value Returns the same result as `land.getAllTowns()`but only for the single nearest town.

Get All Forests

This API returns a list of all essential details of all the forests on the map. Due to the potential size of the response, this API would seldom be called, but would allow exporting of forest information for use in external clients such as `LotATC`, `Liberation` and other mission generators, `DCS-gRPC` clients, online maps etc.

Method signature

```
land.getAllForests()
```

Return value Returns essential information on forest areas on the map. If these forests have a DCS generated ID then they should also be returned.

```
{
    [1] = {
        ["center"] = vec2, -- Calculated center of the shape of the town
        ["shape"] = {      -- List of points that form the shape of the town
            [1] = vec2,
            [2] = vec2,
            [3] = vec2
        }
    },
    [2] = {
        ...
    }
}
```

Get Nearest Forest

This API returns the essential details of the nearest forest to a given vec2 point.

Method signature

```
land.getNearestForest(
    vec2,
)
```

Return value Returns the same result as `land.getAllForests()`but only for the single nearest forest.

Remove Forest

Removes all trees from the specified area. Similar to the remove scenery trigger.

Method signature

```
land.removeTrees(
  point  -- vec2, the center of the area where trees are removed
  radius -- Number, the radius of the area in meters where trees are removed
)
```

Return value None

Get Surface Detailed Data

Modify land.getSurfaceType to return a second value containing any other relevant data on that point. Currently the function returns a value to know if that point is over land, road, runway, and water. The problem is that generally not much other detail is available. For instance "runway" is defined as the whole light grey shaded area on a map which could be taxiways, rampspace, spawn points, the actual runway, etc. More pressingly, the shaded green areas denoting a forest and shaded tan areas denoting a town could also be returned as a quick and dirty check to see if a given point is inside those areas. Those shaded areas are not perfect defintiions, but are generally good enough to be used as a filter.

Method signature

```
land.getSurfaceType(
  point  -- vec2, the point where we want to get the surface type
)
```

Return value

```
Enum surfaceType, enum surfaceDetailedType
```

Get Nearest Clear Area

This API returns the nearest area with no objects in it.

Method signature

```
land.getNearestClearArea(
  point,  -- vec2, the point we want to find the nearest clear area to
  radius -- Number, the required radius of the clear area in meters
)
```

Return value A vec2 of the found point.

Get Nearest Flat Area

This API returns the nearest area with terrain that meets a threshold of flatness.

Method signature

```
land.getFlatArea(
  point,  -- vec2, the point we want to find the nearest clear area to
  radius, -- Number, the required radius of the flat area in meters
  degrees -- The maximum degree of incline from the center of the flat area to the edge
)
```

Return value A vec2 of the found point.

Building Attributes

World Objects and most Static Objects completely lack attribute values. Attributes are table entries used to classify an object and have multiple uses. It is most prevelant with all units and is used by the AI with the searchThenEngage tasks to define what AI is allowed to attack. Currently all map objects either have no attribute or simply have the "Building" attribute. It would be benificial if world objects placed on each map had more detailed attributes.

Having this information would enable a user to write a script that will launch cruise missiles from a ship at multiple points based on attributes in a given zone, for example. Or to reward / penalise players to destroying objects with certain attributes.

If the targets were search radars it is easy enough to get a list of units in a zone, check if each unit has the attribute "SAM SR", and if it does to add that point to a target list. This script would work 10 years from now and with any new SAM added to the game because those units *should* correctly have attributes setup. It isn't telling the script to look for the typeNames associated with each search radar like S-300 SR (Big Bird), S-300 (Clam Shell), Patriot STR, SA-6 STR, etc.

To do the same thing with world objects, lets say "Oil Equipment", the user would need to make a list of object typeNames that correspond with oil equipment. The script wouldn't be valid on another map because different object typeNames could be used and would require the user to create and update the list as maps are released.

Example of search radar unit attributes with a comment stating information that can be derived from the list below.

```
attributes = {
    ["LR SAM"] = true,                  -- It is part of a long range same site
    ["Vehicles"] = true,
    ["SAM elements"] = true,            -- could rely on other units within the group to function as a whole
    ["NonArmoredUnits"] = true,         -- could be damaged by cluster bombs and other lighter muntitions
    ["SAM SR"] = true,                  -- it is a search radar
    ["Air Defence"] = true,
    ["Ground vehicles"] = true,
    ["RADAR_BAND1_FOR_ARM"] = true,
    ["NonAndLightArmoredUnits"] = true,
    ["SAM related"] = true,
    ["All"] = true,
    ["Ground Units"] = true,
    ["CustomAimPoint"] = true,
}
```

Useful attributes for static objects might include such things as

```
attributes = {
    ["building"] = true,       -- Is a building
    ["domicile"] = true,       -- A place where people live (Homes, apartment buildings, barracks)
    ["civilian"] = true,       -- Civilian object
    ["military"] = true,       -- Military object
    ["airfield"] = true,       -- Airfield related object
    ["reinforced"] = true,     -- A reinforced object that requires a penetrating munition to destroy
    ["oil equipment"] = true,  -- Refinery equipment, pipes, storage tanks etc.
    ["storage"] = true,        -- Warehouses, containers
    ["seaport"] = true,        -- objects related to a seaport
    ["bridge"] = true,         -- Road bridge, rail bridge
    ["small"] = true,          -- Small buildings
    ["skyscraper"] = true,     -- Is a skyscraper
    ["industrial"] = true      -- Factories and other industrial objects
    ...
}
```

2. Logistics

Logistics is an oft-requested feature by players. It provides transport helo pilots such as Huey and Hip pilots with another non-combat game-loop.

Servers such as Hoggit have a "logistics lite" game-loop such as deploying and repairing SAM sites or deploying troops. Other servers such as Dynamic DCS allow transport helo players to deploy ground units such as tanks for offensive operations. Blue Flag has its own fuel logistics system whereby slots are blocked if the fuel at an airbase falls below a certain threshold. All of these systems run seperately from the built-in warehouse and airbase inventory system that comes with DCS due to the lack of accessibility.

The built-in warehouse system was created 10 years ago but has seen little use. There have been numerous bugs related to the inventory being corrupted due to weapons being added to the game. The unreliability and opaque functionality has resulted in the system not being used much even though the raw weapon count functionality is better than what can be done via scripts from a purely UI/UX point of view. By opaque functionality we mean the who, what, when, why, and where of supply being sent from one warehouse to another as well as the basic CRUD actions of inventory management.

With script control over warehouses it would be possible for an Mi-8 player to know their team is running low on R-73 missiles at a forward airbase, fly to a warehouse, pickup via slingload or simulated cargo via added weight a cache of R-73 missiles, fly back to a base, and then "deliever" those supplies. The transportation of the supplies is tangible, that player can be shot down, crash on their own, decide to take it to a different base, etc. The current in-game warehouse and resupply functionality is not capable of such gameplay. Servers can then reward players for logistics operations using their own reward system.

As well as the direct player benefits for logistics players, it also allows servers to provide other game-loops such as spawning AI air transports, planes and helicopters, that require escort by other aircraft, or ground convoys that similarly are open to attack and escort. This adds a functional economic warfare to online missions, with the possibility of entire supply chains, which allows players more impactful air-to-ground and escort/intercept operations.

This would provide a huge increase in mission types that can be created by mission scripters in online servers and in order to enable full logistics we do not need APIs that reflect every option in the mission editor.

As well as these new gameplay loops it will make it easier for servers to manage player spawning at spawn points as they can set the inventory to zero and let DCS handle allowing spawning or not instead of scripters having to write their own slotblocking scripts. Finally it will allow PvP servers to more flexibly balance teams via the restrictions of airframes and weapon systems depending on circumstances in the mission.

In order to provide this functionality scripters only need to be able to set inventory on warehouses associated with Airbases, Ships and FARPS. Concepts such as warehouse linking and transfer rates are no longer a factor to consider since this will be in control of the scripters. Therefore this document only covers the minimal APIs to enable this rather than full mission editor parity.

Player spawning changes

Airbase spawns using parking spots, ramps etc. No changes will need to be made to these.

Ground spawns Currently ground spawns are used when there are not enough spawn points in an airbase or when an invisible FARP is being used to create road-bases. Ground spawns are currently not associated with any warehouses so they are always available. Changes would need to be made to associate ground spawns with their nearest warehouse so that spawning can be controlled as with parking spawns. An example implementation would be to treat ground spawns within the rearm radius of a warehouse to be associated with it, otherwise detatch the spawning aircraft and its weapons completely from the warehouse system as with an air-start spawn

Air-start spawns Air-start spawns will be completely detached from any warehouse system for both aircraft and weapons.

New APIs

Get warehouse by name

Get the warehouses associated with Ships, Airbases and Farps where air units can spawn.

Method signature

```
Warehouse.getByName(
  string name -- Name of either Ships, Airbases, or Farps to get their warehouse
)
```

Return value Warehouse object.

Get object associated with the warehouse.

Get the object (Ship, Airbase or FARP) associated with a given warehouse.

Method signature

```
Warehouse.getObject()
```

Return value The associated object (Ship, Airbase or FARP).

Get Warehouse Inventory

Returns the inventory of a warehouse in terms of the weapons and fuel available.

Method signature

```
Warehouse.getInventory(
  category -- Optional Object.Category to limit return values. Realistically only UNIT, WEAPON and maybe Cargo u
)
```

Return value

```
{
    units = {
        [1] = {
            type = 'A-10C', count = 5, max = 12
        }
        [2] = ...
    },
    weapons = {
        [1] = {
            type = 'weapons.missiles.AIM_120C', count = 5, max = 12
        }
        [2] = ...
    }
    fuel = ...
}
```

Set Warehouse Inventory

Set the warehouse inventory of a certain object to a certain value. This is an absolute value which is not an addition or subtraction. Although APIs that add and delete from existing inventory are nice syntactic sugar they are not mandatory as this can be done by using the get/set calls with a minimal chance of race-conditions. Add/Subtract APIs can be layered on top of this method.

Method signature

```
Warehouse.setInventory(
  type,  -- String, the typeName of the inventory to be set. e.g. "A-10C", weapons.missiles.AIM_120C
  count, -- Integer, the number of items
  max    -- The maximum number, this is mainly for UI purposes to show players the maximum allowed at a certain l
)
```

Return value None.

Logistics Events

As well as the APIs above to enable CRUD actions on the warehouse system; implementing the following two events would make it much easier for scripters to know when to call those APIs based on player actions. These two events are only applicable when cargo objects are being slingloaded.

S_EVENT_CARGO_LIFTOFF

Slingloaded cargo object has gone airborne. The location information is ascertained by looking at the unit that landed the cargo.

Fields

```
{
  unit = Unit,    -- aircraft lifting the cargo
  cargo =  Cargo -- The cargo being lifted
}
```

S_EVENT_CARGO_LANDED

Slingloaded cargo has been dropped unharmed at a location. The location information is ascertained by looking at the unit that landed the cargo.

Fields

```
{
  unit = Unit,    -- aircraft landing the cargo
  cargo =  Cargo -- The cargo object being landed
}
```

3. Player specific F10 menu and Output

Currently Eagle Dynamics provides ways to interact with players at the Global, Coalition, Country, and Group levels. However in multiplayer servers where players in the same group are not necessarily working together, this setup lacks granularity to interact with specific players. Servers work around this issue by creating single unit groups, however this then runs into issues whereby group callsigns run out and there is duplication on the server which has knock-on effects. These single unit groups also reduce or remove the benefits of intra-flight datalinks.

Adding player specific F10 radio commands will also allow mission scripters to do things like give certain slots, which can be controlled in terms of who can access them, extra permissions. Therefore the following APIs are proposed to allow scripters and server admins to interact with individual players by extending the existing Global-to-Group APIs to the Unit level.

Out text to player

Display text to an individual player in a unit. See the hoggitworld wiki for the equivalent group example. We will use the `Unit` nomenclature in the APIs to keep consistency with the existing naming scheme even though this is technically for the benefit of players.

Method signature

```
trigger.action.outTextForUnit(
  unitId,      -- integer, the id of the unit. We use ID instead of name here to match
               -- OutTextToGroup which uses groupId
  text,        -- the text to be displayed
  displayTime, -- number, the time the message is displayed in seconds
  clearview    -- boolean, whether it clears the existing text)
)
```

Return value none

Out sound to player

Output a sound file to an individual player. See the hoggitworld wiki for the equivalent group example.

Method signature

```
trigger.action.outSoundForUnit()(
  unitId,    -- integer, the id of the unit. We use ID instead of name here to match
             -- OutTextToGroup which uses groupId
  soundFile -- string, name of the sound file
)
```

Return value none

Add F10 radio command to a player

Add an F10 radio command for an individual player. See the hoggitworld wiki for the equivalent group example.

Method signature

```
trigger.action.addCommandForUnit()(
  unitId, -- integer, the id of the unit. We use ID instead of name here to match
          -- OutTextToGroup which uses groupId
  -- See https://wiki.hoggitworld.com/view/DCS_func_addCommandForGroup for the
  other parameters as they are the same
)
```

Return value Same as addCommandForGroup

Add F10 radio menu to a player

Add an F10 radio menu for an individual player. See the hoggitworld wiki for the equivalent group example.

Method signature

```
trigger.action.addSubMenuForUnit()(
  unitId, -- integer, the id of the unit. We use ID instead of name here to match
          -- OutTextToGroup which uses groupId
  -- See https://wiki.hoggitworld.com/view/DCS_func_addSubMenuForGroup for the
  -- other parameters as they are the same
)
```

Return value Same as addCommandForGroup

Remove F10 radio item from a player

Remove an F10 radio menu for an individual player. See the hoggitworld wiki for the equivalent group example.

Method signature

```
trigger.action.addSubMenuForUnit()(
  unitId, -- integer, the id of the unit. We use ID instead of name here to match
          -- OutTextToGroup which uses groupId
  -- See https://wiki.hoggitworld.com/view/DCS_func_removeItemForGroup for the
  -- other parameters as they are the same
)
```

Return value None.

4. Dynamic Slots

With each module that is released, demands on multiplayer mission slots increase. Even at large airfields, trying to have a reasonable number of slots for each aircraft type exhausts the available parking spots. Additional ground spawns are possible to create, but only within the mission editor itself.

Any mission that has progress across the battlefield requires extra spawns for helicopter modules, and occasionally roadbase-capable modules, for a reasonable player experience. These spawns are generally at FARPs and roadbases which can be created dynamically, but client slots can not be added to them dynamically.

Enabling the dynamic creation and removal of slots for players will open up a wide range of new features. For example players can set up FARPS and road bases to spawn which then opens up further reconnaisance gameplay loops to find these dynamically created bases. Another benefit of allowing dynamic groups is the increased flexibility with which PvP servers can balance out teams as numbers vary during the week.

If the `Client` skill level was honoured when spawning groups via the scripting environment, which resulted in a slot being attached to the units in the group, then this would require no changes to the existing spawn API, it would also enable basic mission planning functionality for things like waypoints and other entries in the spawn table. However a way to remove the slot would be needed which would require a new API.