Megadroid's Mission Mod

Tutorial 1 - Getting Started

Contents

Introduction 3
Structure 3
Prerequisites 4
Visual C++ Runtime 4
Notepad++ (optional) 4
Star Trek Armada 2 4
Installation 5
MMM DSL 5
First Mission 6
The Map 6
Objectives 7
Creating a Script File for our Mission 7
Scripting the Mission 8
Setting up Objectives 8
First Objective: Build a Starbase 10
Second Objective: Build a Shipyard 11
Third Objective: Build 10 Sabres 11
Fourth Objective: Gather 5000 Dilithium 12

Introduction

As you probably already know, MMM is a mod to allow for the creation of scripted single played missions for Star Trek Armada 2. It exposes Armada 2 to Lua to allow for custom missions to be created.

The MMM forum is where most information about MMM can be found. The MMM forum can be found by going to http://www.megadroidsmod.com.

Some knowledge of Lua is required to use MMM. You can learn more about Lua at http://www.lua.org, or by asking in the scripting section of the MMM forum.

Structure

There are three main components to MMM. These, along with their purpose, are described below.

- Lua code
  - All development of MMM missions is done inside Lua. You can call the provided functions in order to manipulate Armada 2 and create your scripted mission. MMM will continue to expose more of the Armada 2 feature set to Lua as it is developed, allowing you to create more detailed missions.
- mmm_loader DSL file.
  - This DSL file is an all-purpose loader for missions. It is a C++ DLL that matches the required signature for an Armada 2 mission script file. When the loader is specified as the mission script it sets up the environment and runs the appropriate Lua file for the mission.
- Armada 2
  - While not technically part of MMM this is where MMM routes all Lua calls. MMM interfaces with the Armada 2 code and uses only the interfaces in the current release of Armada 2. This means that it is compatible with mods (such as FleetOps). There are of course some limitations in Armada 2 that MMM will either protect Lua code from, or apply workarounds to avoid.

Prerequisites

This section details programs and redistributables that need to be installed before you will be able to create missions with MMM.

Visual C++ Runtime

MMM is built using the 2010 Visual C++ runtime and requires that it be installed on the target computer. You can download the redistributable from the Microsoft website, at http://www.microsoft.com/downloads/details.aspx?FamilyID=a7b7a05e-6de6-4d3a-a423-37bf0912db84. End-users will also need to install this redistributable before they can play any missions with MMM.[1]

Notepad++ (optional)

Although not required, Notepad++ has syntax highlighting support for Lua, the language that missions are scripted in with MMM. This makes it much easier to develop and read mission scripts. Of course, any other text editor will work if you have another preference. You can download Notepad++ from http://notepad-plus-plus.org/.

Star Trek Armada 2

Obviously you will need an installation of Armada 2 with either the latest official patch or the latest release from the Armada 2 Patch Project - both of which can be obtained from the Fleet Operations website at http://www.fleetops.net/. MMM does not modify this installation (unless you replace the campaign maps), and so you can continue to use the installation as before. MMM is also compatible with Fleet Operations[2], so you can develop missions for it too.

[1] This may be changed in the future if this is deemed to be a problem.
[2] There are some issues with ship reference persistence that are being worked on.

Installation

Armada 2 uses DSL files for custom mission scripts. A DSL file is simply a renamed DLL file - a dynamically linked extension to an application. MMM is contained inside a DSL file and is loaded by Armada 2 when it is specified as the script file for a map.

MMM DSL

Download the mmm_loader DSL file from the releases thread in the news section of the MMM forums. Make sure to download the latest version - this will be the one with the latest date. There is also a DSL included with this tutorial that you can use, but there may be a newer version available.

Once you have downloaded the DSL file you need to copy it into your `Armada 2/missions` directory. This is where Armada looks for DSL files when it loads a map.

That's it for installation - now we can move on to making a basic map to get started using MMM. This map will be used in some future tutorials as a base starting point, so it would be handy if you kept a copy lying around to refer back to.

First Mission

To get started we will make a very basic map and a very basic mission to go along with it. We will be using the empty framework provided on the MMM forum for this tutorial since it defines the correct structure. We will be calling this mission "First Mission".

The name of the mission is important; the filename of the script file for the mission must be the same as the title of the map.

The following are the objectives for player in this simple mission:

- Build a starbase
- Build a shipyard
- Construct 10 ships (sabres)
- Gather 5000 dilithium

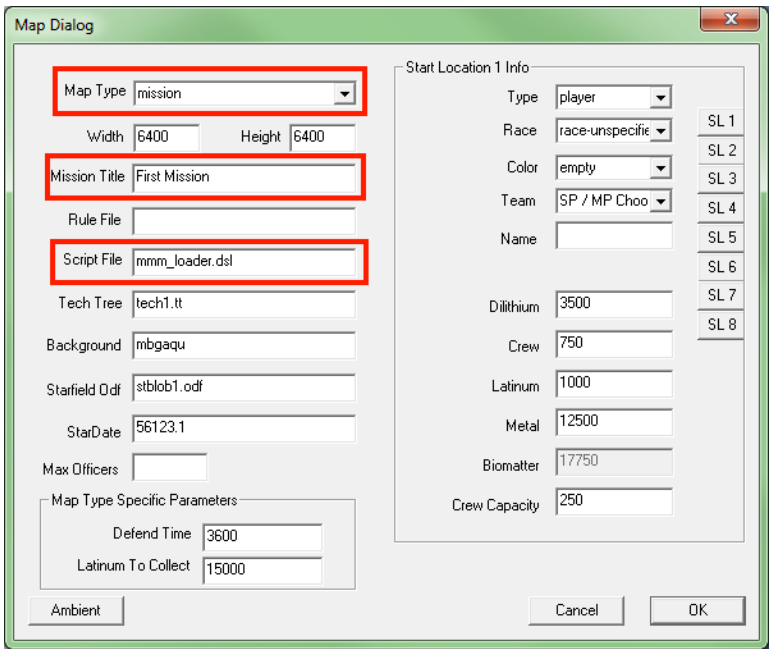
The Map

The map for this mission will be very basic - simply a constructor and a few resources to gather from – a finished version of the map is included if you don't want to make it yourself.

It doesn't really matter what you do for the map as long as these settings are set.

- The map type should be set to *mission* (MMM can technically run in any map type as they all allow for the setting of a script file, but single player missions are all of the *mission* type).
- The mission title needs to be set to something - the .lua file used later for script will be [*Mission Title]*.lua, where *[Mission Title]* is whatever you put in the box.
- The script file must be the MMM DSL. You can rename the DSL if you want to, but the default is to use mmm_loader.dsl.

The screenshot below shows the map settings dialog with the correct settings.



Once you have finished setting up the map save it in the bzn folder as *a2_tutorial_ui.bzn*. This will replace the existing 1st tutorial mission. We do this because it allows us to access it via the menu easily without altering other files. A future tutorial will deal with custom campaign display.

Objectives

You will recall that the objectives for the player were:

- Build a starbase
- Build a shipyard
- Construct 10 ships (sabres)

For the player to see these objectives we need to make an objectives file which we will load in the mission script later on in the tutorial. To create objectives that can be manipulated in script, an objectives file must be properly formed. Our objectives file looks like this:

```
First Mission

To ensure the continued safety of this area you have been ordered to build a base and a small fleet of ships.

*Build a Starbase
*Build a shipyard
*Build 10 Sabres
*Gather 5000 dilithium
```

The first line is interpreted as a heading (the title of the mission). An objective is denoted with a * prefix. Armada will interpret this as such and create a checkbox on the objectives screen. You can then use the MMM script interface to check or uncheck this box as the mission progresses. Anything without a * prefix is simply displayed as normal text.

Name this file *first_mission.txt* and place it in the `Armada 2/objectives` folder.[3]

Creating a Script File for our Mission

You can download the starter script file (EmptyFramework.lua) either from the MMMApp framework thread in the Tutorials forum on the MMM board, or you can get it directly from here. Put this file in the `Armada 2/missions` directory (in the same place as the mmm_loader DSL). Since our mission is going to be called "First Mission" we need to rename the EmptyFramework.lua file to "First Mission.lua".

You can read more about the MMMApp framework and how it is used on the forum.

At this point you should have the following files you should have changed or added in your Armada 2 directory.

```
Armada 2
       missions
              mmm_loader.dsl
              First Mission.lua
       objectives
              first_mission.txt
       bzn
              a2_tutorial_ui.bzn
```

Now that we have all the files we need we can start to add to our mission script.

[3] If you have already started Armada, you will need to restart it in order for it to pick up the new objectives file.

Scripting the Mission

Setting up Objectives

Objective Display

First, let's get our objective text loaded into Armada and displaying to the player. You should have already created the objectives text file and placed it in the objectives folder. In order to get these objectives into the Armada 2 objectives display we will have to call two MMM functions, both of which are part of the *Mission* library. These are *loadObjectives* which loads objectives from a file and *showObjectives* which can be used to either show or hide the objectives display.

We will need to pass the filename of our objectives file (which is first_mission.txt) to *Mission.loadObjectives* and the boolean value *true* to *Mission.showObjectives* to make the objectives display appear. Go to the function *TestMission:init* and make the following changes. The lines you need to add are highlighted.

```
function TestMission:init( )

        Mission.loadObjectives( "first_mission.txt" )
        Mission.showObjectives( true )

end
```

If you play the mission now (go to the single player menu in Armada 2 and run the first tutorial mission) you should see the objectives window appear with our text in it. If not, there will most likely be an MMM error being displayed in the top right hand corner of the screen which should point you in the direction of the problem.

Now that we have the display working we can move on to the back end of objective - the bit that actually does all the work.

Objectives Code

The flow of our mission script logic is based around objectives, so it is important for us to organise these in a good way. We will make use of a simple lookup table which will map identifiers to boolean values, but you can use something more complex in the future if the situation requires it.

To start with we will need some way of identifying each objective. MMM uses 0-based indexing to refer to the objectives in the objective display, so we will do the same. Add the following lines just above *TestMission.new* (near the top of the file).

```
--Some constant values for lookups
Objective_Starbase  = 0
Objective_Shipyard  = 1
Objective_Ships     = 2
Objective_Dilithium = 3
```

In effect this creates aliases for these numbers; instead of saying "set objective 0 to true" we can now say "set Objective_Shipyard to true". This helps with readability and also makes changing the inner value easy - you only have to change it in one place instead of hunting through the code.

Now that we have defined the indexing constants we need to create the table that we will store the objective data in. Edit *TestMission.new* so that it looks something like the following. Again, new lines are highlighted.

```
function TestMission.new( )
        local newMission = { }
        setmetatable( newMission, TestMission.mt )

        --You can set up properties for the mission here, or you can do it in init.

        --This creates a new lookup table.
        newMission.objectives = {}

        return newMission
end
```

This line simply creates a new table and stores it in the newMission instance that we are creating. It will be accessible through self from this point on. If you haven't already you should read up on tables in the official Lua documentation.

We can use our constants as indices into this new table; we will store boolean values representing the state of the objective, with false representing an incomplete objective and true being a completed objective. We can add code to initialise our objectives into the *TestMission:init* function, as shown below.

```
function TestMission:init( )

        Mission.loadObjectives( "first_mission.txt" )
        Mission.showObjectives( true )

        --Here we will initialise our objectives
        self.objectives[Objective_Starbase]   = false
        self.objectives[Objective_Shipyard]   = false
        self.objectives[Objective_Ships]      = false
        self.objectives[Objective_Dilithium] = false

end
```

This sets the value of each of our objectives to false. If we want to mark an objective as complete, we would use *true* instead.

Now although we can now set objectives to be true or false the objectives display will not be automatically updated. We have to call *Mission.setObjective* to do this. We will create a separate function to do this and call it from *TestMission:update*. The following snippet shows the changes needed.

```
function TestMission:update( )
    self:synchroniseObjectivesDisplay ( )
end

function TestMission:synchroniseObjectivesDisplay( )
        for index, value in pairs( self.objectives ) do
                Mission.setObjective( index, value )
        end
end
```

While this might look complicated at first it is reasonably simple once you break down what it is doing. Every time the mission script is updated, we call the function *synchroniseObjectivesDisplay*. This function takes advantage of how the table is laid out to neatly update the objectives display.

The table looks like this:

Index	Value
Objective_Starbase	false
Objective_Shipyard	false
Objective_Ships	false
Objective_Dilithium	false

The instruction *for index, value in pairs(self.objectives)* used in the synchronise function requests an index-value pair from self.objectives until there are no more pairs left to get. For each of these pairs it sets the objective number represented by *Index* to the value stored in *Value*. So the first pair retrieved is:

Index : Objective_Starbase
Value : false

So calling:

Mission.setObjective(index, value)

effectively does:

Mission.setObjective(Objective_Starbase, false)

As you will remember, Objective_Starbase has the value 0, so this finally resolves to:

Mission.setObjective(0, false)

which sets the 0th objective to false.

Running the mission now should do the same thing - the objectives display will be shown. Try setting one of the values to *true* in the code inside *TestMission:init* to see the effect. If you did it right, the appropriate box will be ticked.

First Objective: Build a Starbase

Now that we have the objectives structure implemented we can start to add code to deal with each of the objectives. The first 3 objectives are implemented quite similarly in that they all examine the player team's entities.

Let's get on with the first objective: building a Starbase. To check this objective we will have to examine the entities that the player has and look for one with the odf "fbase.odf". The following code does that, using the *in pairs* construct that we saw earlier.

```
function TestMission:update( )
    self:checkStarbaseObjective( )
    self:synchroniseObjectivesDisplay( )
end

function TestMission:checkStarbaseObjective( )
        if not self.objectives[Objective_Starbase] then
                local ents = Team.new( 1 ):getEntities( )
                for  , v in pairs( ents ) do
                        if v:getOdfName() == "fbase.odf" then
                                self.objectives[Objective_Starbase] = true
                                Mission.showObjectives( true )
                                break
                        end
                end
        end
end
```

The code does what was said above - it looks at each one of the entities that the player team owns and checks the ODF name. If it finds one with the starbase odf (*"fbase.odf"*) then it completes the objective (by setting it to true) and shows the objective screen. The break statement means that the

code stops checking at that point. All of the code in the function will only execute if the starbase objective has not already been fulfilled, which is what the outermost if statement is for.

If you run the mission now and build a starbase an objective should be completed and the objectives display will be shown.

Second Objective: Build a Shipyard

The second objective is quite similar to the first, simply looking for a different ODF name. As such it does not require much explanation. The definition for this function follows.

```
function TestMission:update( )
    self:checkStarbaseObjective( )
    self:checkShipyardObjective( )
    self:synchroniseObjectivesDisplay( )
end

function TestMission:checkShipyardObjective( )
        if not self.objectives[Objective_Shipyard] then
                local ents = Team.new( 1 ):getEntities( )
                for  , v in pairs( ents ) do
                        if v:getOdfName() == "fyard.odf" then
                                self.objectives[Objective_Shipyard] = true
                                Mission.showObjectives( true )
                                break
                        end
                end
        end
end
```

As you can see the only differences are that we are checking and setting Objective_Shipyard instead of Objective_Starbase and we are checking for *fyard.odf* instead of *fbase.odf*.

Third Objective: Build 10 Sabres

The third objective is also quite similar except with the addition of some counting. We iterate over all the ships that the player owns and check to see if it is a Sabre (*fdestroy2.odf*). If it is, we add to the count of Sabes (*shipCount*). If *shipCount* is greater than or equal to 10 we mark the objective as completed.

```
function TestMission:update( )
    self:checkStarbaseObjective( )
    self:checkShipyardObjective( )
    self:checkShipsObjective( )
    self:synchroniseObjectivesDisplay( )
end

function TestMission:checkShipsObjective( )
        if not self.objectives[Objective_Ships] then
                local ents = Team.new( 1 ):getEntities( )
                local shipCount = 0
                for  , v in pairs( ents ) do
                        if v:getOdfName() == "fdestroy2.odf" then
                                shipCount = shipCount + 1
                        end
                end

                if shipCount >= 10 then
                        self.objectives[Objective Ships] = true
                        Mission.showObjectives( true )
                end
        end
end
```

Fourth Objective: Gather 5000 Dilithium

The final objective is different to the other objectives but is simpler to implement. We are simply going to check the resource levels of the team. If the resources are greater than 5000, we mark the objective as completed. The *Team* library and instances created of it have many different functions to query team information.

```
function TestMission:update( )
   self:checkStarbaseObjective( )
   self:checkShipyardObjective( )
   self:checkShipsObjective( )
   self:checkDilithiumObjective( )
   self:synchroniseObjectivesDisplay( )
end

function TestMission:checkDilithiumObjective( )
       if not self.objectives[Objective_Dilithium] then
              local team = Team.new( 1 )
              local amount = team:getResource( Resource.Dilithium )
              if amount > 5000 then
                     self.objectives[Objective_Dilithium] = true
                     Mission.showObjectives( true )
              end
       end
end
```

If you run the mission now you should be able to play through and achieve all the objectives. This concludes this basic tutorial for MMM. More tutorials will be available on the MMM forum, where you can also ask questions about this tutorial or any other part of MMM development.