NEXUS

The Jupiter Incident

Basic Mod Creation

Tutorials written by Arparso and Jusas

Re-written and edited by The Old Dragon

Foreword

Hello and welcome to this tutorial which, by the end of, we hope you'll be able to begin writing your own mods from scratch. As you can probably tell by the title, this document is aimed at those looking to begin modding Nexus or those just beginning who, either case, find themselves floundering amongst the structures, commands and weird names of the Nexus scripting language… so don't worry, help has arrived.

When I first started trying to learn this myself a little over a year ago, I was lucky enough to run into a couple of people who'd been around and find some tutorials. Those tutorials have however disappeared over time, so I've decided to re-edit them for a new generation of Nexus modders.

So, without further ado, on with the aims and the actual tutorial (instead of having to read through my ramblings!). What I'm aiming to do here is…

1) Express how the Nexus script is set out.
2) How to set up the file structure to allow your mod to work.
3) Create solar systems from scratch.
4) Write your first mission.

Now, I'll hand you over to Arparso and his tutorial on the Nexus scripting language.

****

NEXUS scripting language

MACHINE TUTORIAL

This tutorial is meant for beginners. It covers the basics of how MACHINES and their STATES work and how they can be used for full effect in your own mission.

What is a MACHINE?

The MACHINE consists of STATES and these consist of RULES.

The term “machine” fits these constructs surprisingly well, because they work just like most machines you would encounter in real life. Like a ticket automat for example: normally he waits for a new customer. We can say, he is in an “idle” STATE. If someone comes along and presses some keys, the automat changes to an “awaiting money” STATE. Then the user inserts some coins and the automat begins processing the input, prints the ticket and switches back to the “idle” STATE again. Now he waits for the next customer…

MACHINES just work like this. They can have multiple STATES, of which only one is active at the same time. The RULES give the MACHINE exact guidelines about what he has to do in a given STATE… for example, how he should process the given input or what conditions he has to check.

Basic Structure

The basic Syntax is as follows:

```
MACHINE “name”
     STATE <name>
          [RULE event <event-name>
               ...
     END]
     END
END
```

What happens here?

MACHINE “name”	=> defines the MACHINE’s name
STATE <name>	=> defines a STATE’s name
RULE event <event-name>	=> defines a RULE event

…   => here is the place, where all your commands will go

MACHINE, STATE and RULE each have their own “block”, which is ended with the keyword END. That keyword tells Nexus, that this RULE, STATE or MACHINE is finished now.

Lets take a closer look at each block separately

The RULE block

The basic syntax of the RULE block:

```
RULE event <event-name>
      [:condition
      Condition1
      Condition2
      ...
      :end]
:action
      [Command1]
      [Command2]
      ...
:end
END
```

Every RULE follows this syntax. First Nexus will look into the “:condition” block and see if all defined conditions are met. If they are not, Nexus won’t run any of the commands in the “:action” block. If they are met, then Nexus will continue with the “:action” block.

Hint: You can leave out the whole “:condition” block, if you don’t want to set any conditions.

Hint: Instead of “:condition … :end” you can write “condition condition”, if you have only one or a few simple conditions to check. You can link conditions with & (AND) and | (OR).

The ":action" block is the part of the RULE, where all your commands go. You can give orders to ship, script cutscenes, assign variables and a lot of other stuff here. A more detailed explanation can't be given here, because this would exceed this tutorial's scope by far. Look at the provided Mod Documentation by Mithis for more details.

Some quick examples for RULES:

```
RULE event display_title
:action
      Title(0,0,0,"Mission completed");
:end
END

RULE event do_nothing
:action
:end
END

RULE event process_condition
      :condition
            raven.damage>100
            raven.race=#race_Gorg
      :end
      :action
            Title(0,0,0,"Mission completed");
      :end
END

RULE event short_condition
      Condition a>b
      :action
            Debug("A is bigger than B");
      :end
END
```

Naming conventions: Basically you can set about every name you can think of. But keep in mind, that Nexus knows three predefined RULES: In, Out, Tick.

Every STATE needs a “RULE event In”. This RULE is being run, when the MACHINE reaches a given STATE. Without the In event, Nexus won’t know, which of the multiple RULES in a single STATE it should run first, so make sure, you have the In event.

The Out event is not needed at all, but it’s always good to create a proper “end” RULE. That would be the RULE, where you leave the current STATE.

The Tick event is a very special event. This event runs every few moments and doesn’t need to be called by another event. You need to write “TICK <time_in_seconds>” somewhere between the RULES blocks. This way you can define, how often the Tick event is run. TICK 5 runs it every 5 seconds.

Tick is great for monitoring certain events. Tick could check every few seconds, if ship X has lost his shield for example. Based on the result, it could trigger another RULE to be run. Here is an example:

```
TICK 5


RULE  event tick

      :action

            If(a>b, a_bigger_b);

      :end

END


RULE event a_bigger_b

      :action

            DEBUG(“A is bigger than B”);

      :end

END
```

As you see, a STATE can have multiple RULES of course. You can even have multiple RULES with the same name! Something like this for example:

```
RULE event ab_compare

      Condition a>b

      :action

            DEBUG(“A is bigger than B”);

      :end

END
```

```
RULE event ab_compare
      Condition a<b
      :action
            DEBUG("A is smaller than B");
      :end
END
```

Calling a RULE from within a RULE: Now how do you do that? Simple:

```
LocalEvent(name_of_event);

or even

name_of_event;
```

Note, that you can only call a RULE that is given in the current STATE. You can't call RULES from different STATES that way.

The STATE block

The basic syntax of the STATE block:

```
STATE <name>
[RULE event <event-name>
      ...
END]
...
END
```

There is nothing special to be seen here. You define the different STATES of your MACHINE given the above syntax. Don't forget to choose appropriate names for your STATES (and RULES, too), so you can easily see, what happens in this STATE or RULE.

You can have as much RULES as you want in a single STATE as you can have as much STATES as you like in a single MACHINE. It's up to you to decide, where you could use a separate STATE and where it is sufficient to define a few more rules in an already existing STATE. Do whatever you like more, but try to create a clear structure of MACHINES, STATES and RULES, so that other people have a chance to understand, what your code achieves.

> **Hint:** Don't forget to give every STATE his own "In" RULE. If you switch to another STATE, the program will run this RULE at first.

How to switch to another STATE: Simple… use:

```
ChangeState(name_of_state, parameter);
```

With this you can change only to a STATE in this MACHINE. To change the STATE of a different MACHINE, use:

```
M:GetMachine(name_of_machine):ChangeState(name_of_state, parameter);
```

With "parameter" you can give local variables to another STATE. If you don't want to do this, just write 0 (zero) instead of the variables.

Here's an example for two simple STATES in a MACHINE:

```
MACHINE "ab_compare"

STATE a_bigger_b

    RULE event In
        :action
            Debug("A is bigger than B");
        :end
    END

    TICK 5

    RULE event tick
        :action
            If(a<b, ChangeState(a_smaller_b,0));
        :end
    END
```

```
END

STATE a_smaller_b

      RULE event In
            :action
                  Debug("A is smaller than B");
            :end
      END

      TICK 5

      RULE event tick
            :action
                  If(a>b, ChangeState(a_bigger_b,0);
            :end
      END
END
END
```

The MACHINE block

The basic Syntax is as follows:

```
MACHINE "name"

      STATE <name>

            [RULE event <event-name>
                    ...
      END]

            ...

      END

      ...

END
```

Now you know almost every important aspect of a MACHINE's structure. Here are some additional bits of information as well as some repetition of important facts you learned throughout the tutorial:

Multiple RULES: You can define multiple RULES per STATE, but you only have access to the RULES in the currently active STATE. RULES can be run simultaneously.

Multiple STATES: You can define multiple STATES per MACHINE. You have access to all the RULES defined for that STATE. STATES can't be run simultaneously – only one STATE can be active at a time. However, you can switch between the STATES as you like.

Multiple MACHINES: One thing, that wasn't covered yet. You can define multiple MACHINES in a mission. These run all at the same time. This way you can create totally different MACHINES for totally different purposes. For example you create one MACHINE for the enemy AI, another MACHINE, that constantly checks, if the winning conditions have been already met and a third MACHINE, that controls the in-game cutscenes.

Activating MACHINES: You do that best in the "sceneInit" RULE of your mission with the command "M:GetMachine(name_of_machine):ChangeState(name_of_state, parameter);

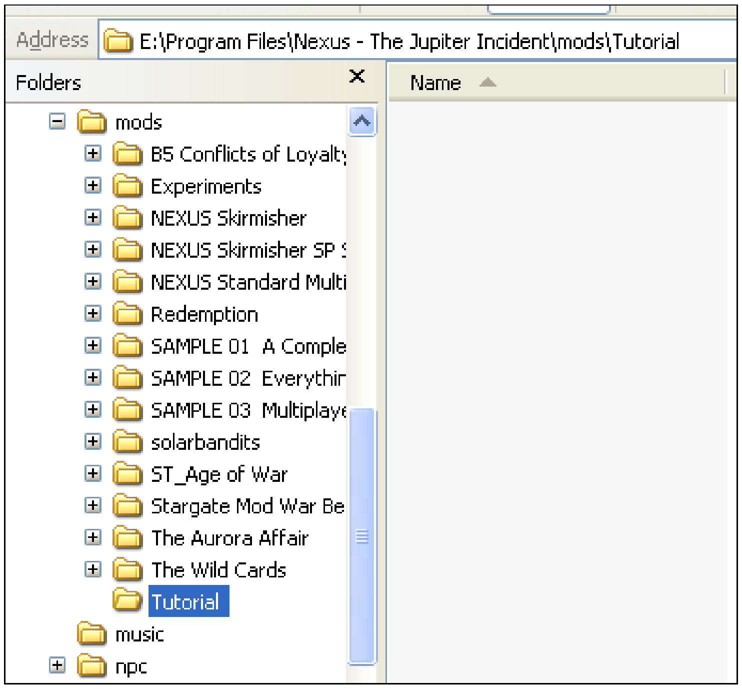
Closing words

I wish you luck and much fun while trying to create new mission and mods for Nexus. Hopefully this tutorial will help you understand the basics of MACHINE scripting and get things rolling for you.

****

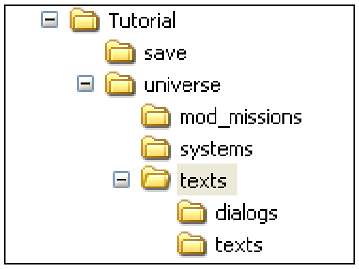
And thank you very much Arparso, now it's time for my hard work to begin (lots and lots of typing…), so what's next? Ah yes, the directory structure of the basic mod…

Ok, so the very first thing we're going to do is navigate our way to the Universe/Mods directory, where we're going to create a new folder called (rather imaginatively) "Tutorial". So we should now have something like…



Ok…so now we'll enter our new 'Tutorial' folder and add another, this one we'll call '**universe**'. Then we'll go into our new folder and create some more folders, these we'll call '**mod_missions'**, '**systems**' and '**texts**' respectively. Finally, we have one more folder to add which will enable our ship captains to insult each other during the mission. So go into the '**texts**' folder and create two more, called '**dialogs**' and '**texts**'.

Hopefully, we should now have a directory structure looking something like this…



And that folks, is all you need to actually get a mod to run. Obviously, just by looking at the Nexus game itself or any of the mods already done, there are more folders in play… but these are for more advanced parts of the game that we won't be covering in this tutorial. So there's no need to confuse anyone with lots of things that we won't be using.

Right then, we've completed the file structure, so now it's time create a location to stage our mission in. Welcome to… well, part three I guess you could call this, how to create solar systems.

****

Creating Solar Systems

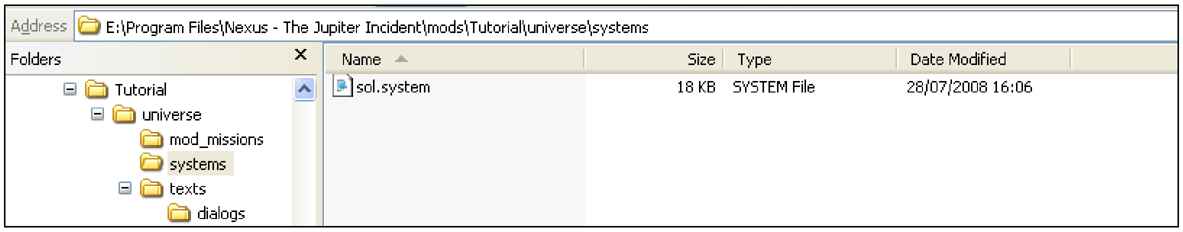
In this part, we going to learn how to create a new solar system from scratch. Now we could simply use a system that someone's already created, but there's no fun in that and we are here to learn after all (Besides, it's part of the **mission making** tutorial that's coming up next anyway). I'd also just like to take a moment to say thank you to **Jusas**, for writing the tutorial that I'm using as a foundation for this part.

Believe it or not, making a new system is quite simple, just a little time consuming and if you try reading the manual through first, then it can seem quite daunting. Hopefully, this version will be much easier on the eye (and mind!).

Now, before we begin, I'd just like to point out that I do my scripting with **notepad**, but you're free to use whatever's comfortable for you. So let's begin…

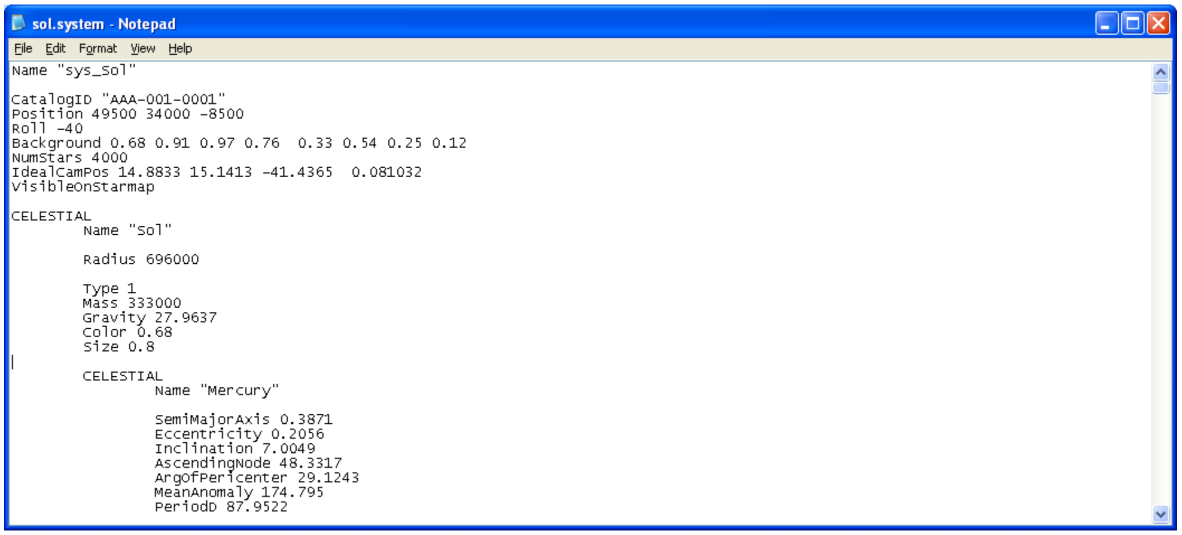
Our first step is actually a bit of cheating, what we're going to do is navigate our way through the original campaign files by this path – '**Nexus - The Jupiter Incident\universe\systems**', where we should find the file '**sol.system**'. Copy this file and place it in the new '**systems**' file we just created.

So we should be looking at…



Name	Size	Type	Date Modified
sol.system	18 KB	SYSTEM File	28/07/2008 16:06

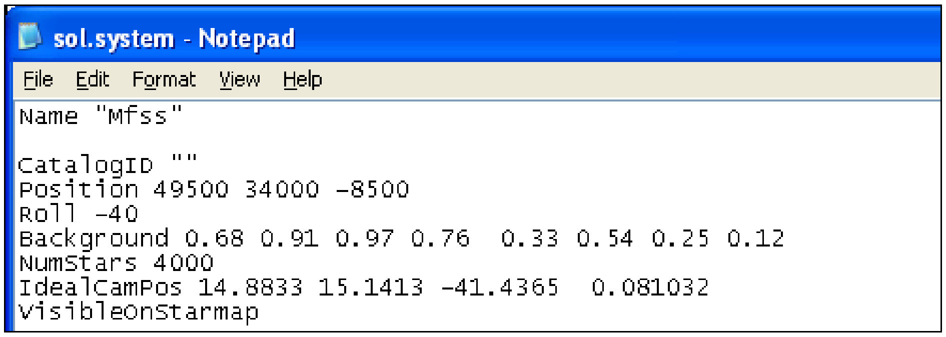
Open the **sol.system** file and you should find yourself looking at this…



```
Name "sys_Sol"

CatalogID "AAA-001-0001"
Position 49500 34000 -8500
Roll -40
Background 0.68 0.91 0.97 0.76  0.33 0.54 0.25 0.12
NumStars 4000
IdealCamPos 14.8833 15.1413 -41.4365  0.081032
VisibleOnStarmap

CELESTIAL
        Name "Sol"

        Radius 696000

        Type 1
        Mass 333000
        Gravity 27.9637
        Color 0.68
        Size 0.8

        CELESTIAL
                Name "Mercury"

                SemiMajorAxis 0.3871
                Eccentricity 0.2056
                Inclination 7.0049
                AscendingNode 48.3317
                ArgOfPericenter 29.1243
                MeanAnomaly 174.795
                PeriodD 87.9522
```

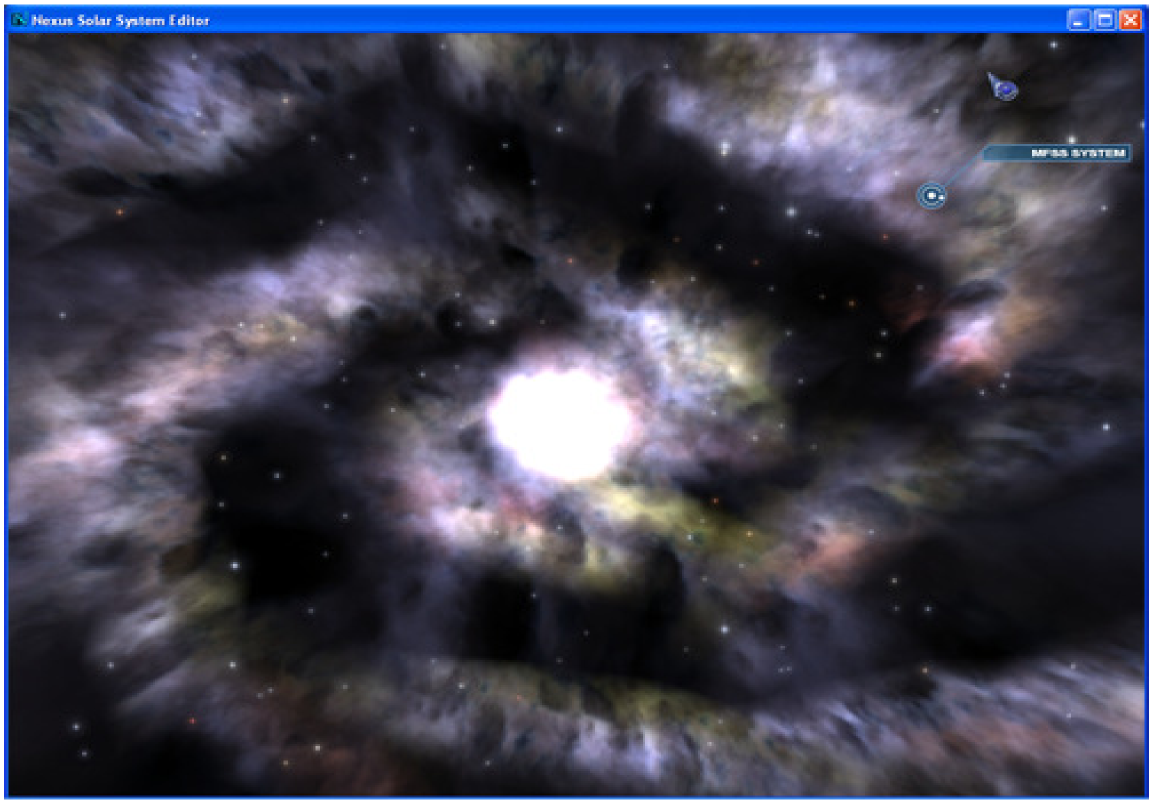
Now all we're really interested in is the first few lines, everything below the first celestial line (and including) can be deleted safely. So let's get rid of the unwanted stuff and change the name to "My First Solar System", but as that's a bit of a mouthful, we'll abbreviate it to 'Mfss' (oh, and don't be tempted to add '**system**' to the name, the editor will do this for you!). What we'll also do, is 'clear' the CatalogID field too (I'm not quite sure what this is for to be honest, but it doesn't seem to affect anything).

So…



```
Name "Mfss"

CatalogID ""
Position 49500 34000 -8500
Roll -40
Background 0.68 0.91 0.97 0.76  0.33 0.54 0.25 0.12
NumStars 4000
IdealCamPos 14.8833 15.1413 -41.4365  0.081032
VisibleOnStarmap
```

This is what we should now be looking at. Believe it or not, that's about all we need to do with this file, from now on everything we need to do is handled by the **Solar System Editor**. The only thing we may be required to play around with is the system position, so I'll bring up a picture of the map you'll soon be looking at…



Ok, now we have this, if you take a look at the **.system** file you've created, you'll see that the third line of text holds the galactic position data. The first thing to know is that '**position 0 0 0**' is the centre of the map. So with that in mind…

1) A positive first number will push us '**east**' while a negative number will pull us '**west**'.

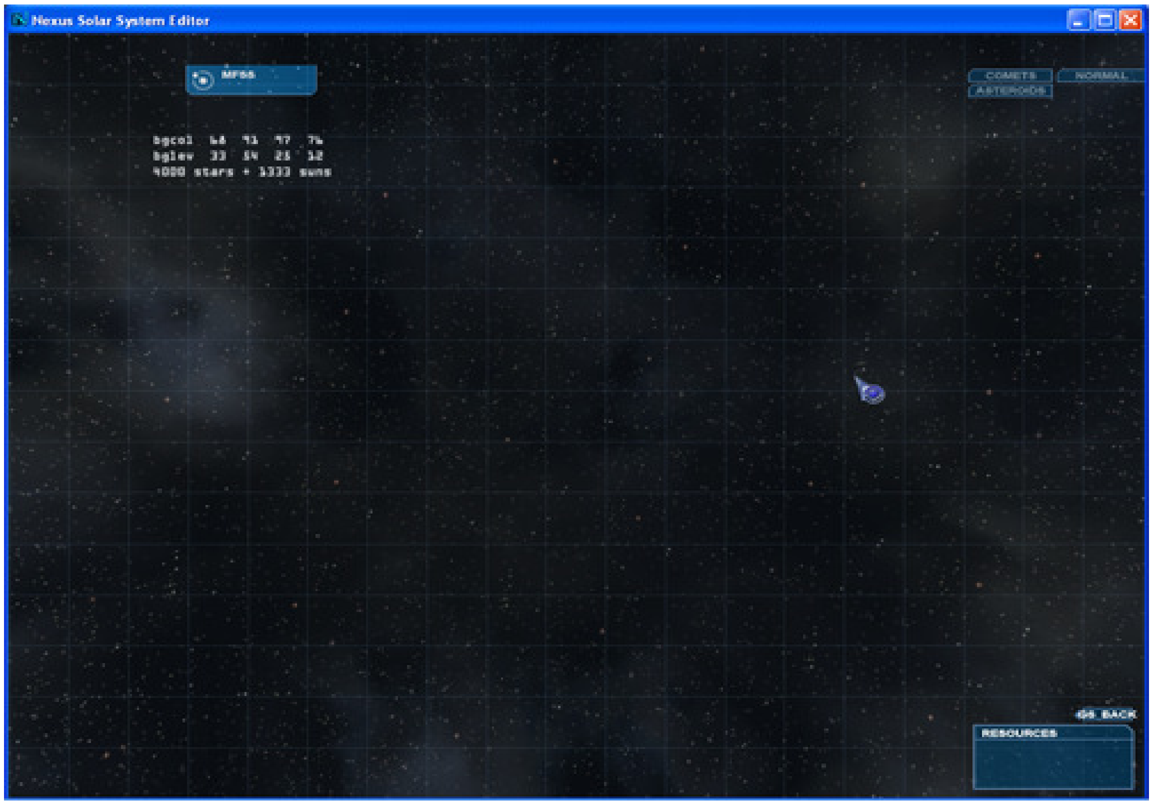
2) A positive second number will send us '**north**' and the negative will take us '**south**'.

3) As I can't really see any difference, I'll hazard a guess that a positive number will '**raise**' our systems position and a negative will have the opposite effect (not sure what effect this has in the game, but there we go).

If you wish to, feel free to play around with the positional data and when you're ready, save and close the file. It'd also be wise to point out that you must save your text files (not necessarily close them) before starting up either of the editors or they will <u>not</u> take any notice of your recent work! You may consider it obvious, but I've done it several times and sat there wondering why there's been no changes in the editor, so be warned.

One last thing before we fire up the editor, rename your **.system** file to '**Mfss.sys'** for both your benefit and the games (let's face it, we're going to get confused enough in our modding careers so why add to it unnecessarily?).

Ok, let's start up our system editor! Run the **mod_tools.exe** from the **Nexus\mod_tools** folder, select our '**Tutorial**' from the drop down menu and click on the **Solar System Editor**. This should now open up the galactic map very similar to the one above with our system somewhere on it (all depends as to whether or not you tinkered with it), double click on it to go into the system view (click again if you get bored with the scenic view) and you'll be greeted with…



```
bgcol  68  91  97  76
bglev  33  54  25  12
4000 stars + 1333 suns
```

Bit boring huh? So let's brighten it up… the first thing we're going to do is add a sun with the following key combination –

CTRL-SHIFT-U

Aargh!! Bright lights, bright lights!

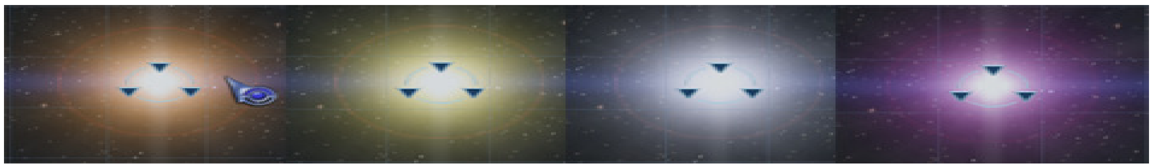
Who let those blooming gremlins in here again?!

But they do have a point though, the sun we've just created may not be just what we want. So let's change it, by using **Numpad +** and **–** we change the size or intensity of the sun (depends on your point of view). And with a playing, we can alter the colour (or temperature if you prefer) with **Numpad /** and *****.

So go have a play, I'll still be here when you get back. Then we'll progress to planet creating (isn't it fun being god of your own universe?).

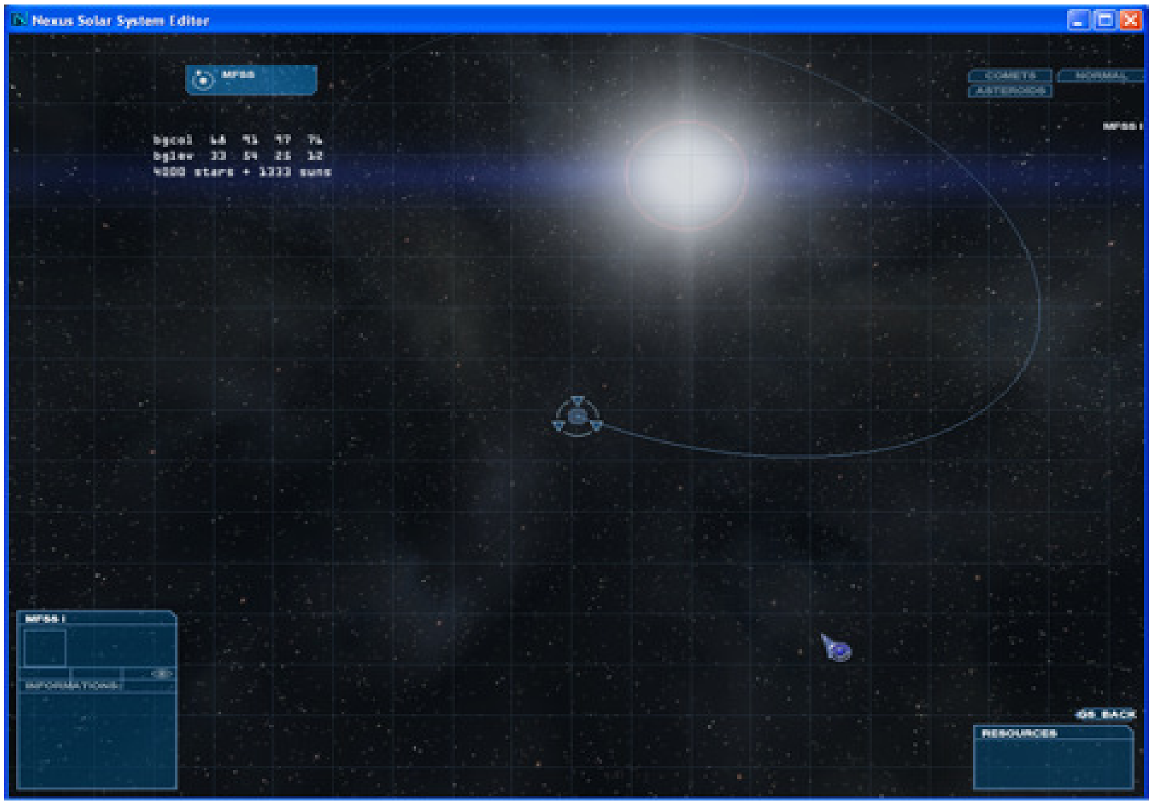
Finished…? No…? okey dokey.

Da dee dum dum da, da deed um.. well, here's a few of mine I did while I was waiting…



A-ha, your back! Let's continue then…

What we're going to do is create a small moon without an atmosphere in close orbit to our sun. Sound difficult? Don't worry, it's easy peasy. First of all, click on the sun to select it, then press **CTRL+SHIFT+C**… and hey presto, we have a randomly generated planetoid!



```
bgcol  68  91  97  76
bglev  33  54  25  12
4000 stars + 1333 suns
```

Now, whenever we create something new in here, the camera will automatically zoom in on it and the item will become our selected item.

There’s also a lot we can do with the planetoid too, so let’s do some experimenting…

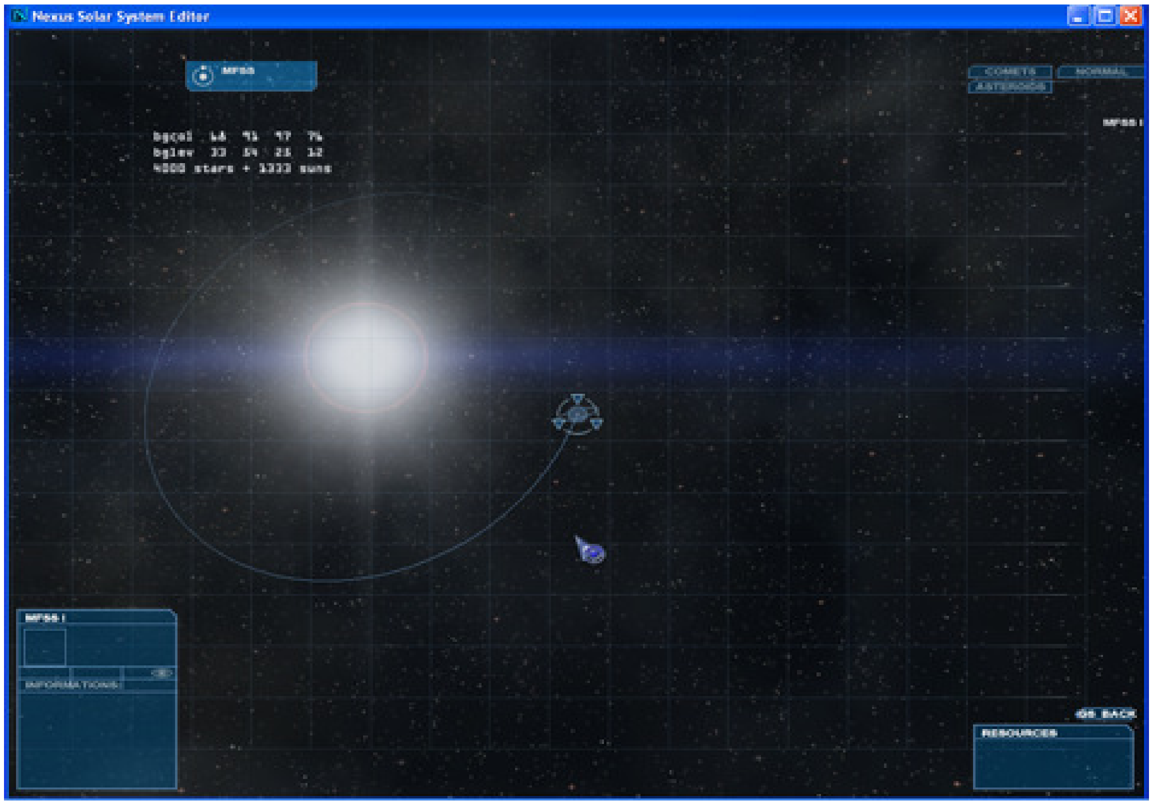
Just say we don’t like the look of the planetoid… by pressing **SHIFT**+**C** (repeatedly if we chose too) we can change its size and appearance to another randomly generated one. Or if desired, we can take or more authorative approach to editing.

We can change the size with:	**Numpad +** and **–**
It’s colour and texture with:	**Numpad 4** and **6**
Increase its’glossiness’ (R,G,B spectrum) with :	**Numpad 1, 2** and **3**
Or decrease it with:	**Numpad 1, 2** or **3**

We can also add some planetary rings to it as well (a barren lump of rock with planetary rings may not look quite right, but hey… we’re learning here!). To do this, we use the following commands…

Rings on, cycle through the different textures, then off:	**R**
Adjust the outer radius:	**Home** and **End**
Adjust the inner radius:	**CTRL** + **Home** and **End**
Adjust the rings opacity (How ‘see through’ we want it):	**SHIFT** + **Home** and **End**

So go ahead and have a play with those, get the hang of making the planet you’d like. Then we’ll get onto learning how to adjust its orbit, as a planets position can have quite an aesthetic appearance in the mission.



See the ring around the sun? As you've probably guessed, this is our planets 'orbital path' and as with everything else so far, we're going to tinker with it… We can move the planetoid forward along its path with the **Right** arrow key and backwards with the **Left** arrow key. We can also change the angle of the orbit too, with the **Up** and **Down** arrows. After all, five or six planets wouldn't look quite right if they were all in a straight line, would they? The one other thing we can do with orbital paths, is change the shape of the orbit, making it more elliptical. We can do this with the **CTRL** + **PG UP** and **CTRL** + **PG DN** keys.

Before I forget and drift off to another topic, I'd best tell you about some of the other objects we can create. As you now know, by pressing **CTRL**+**SHIFT**+**C**, we create a random non-habitable planetoid with no atmosphere. But if we change the letter to one of the following, then we change what we're creating…

A	Habitable plant/moon
B	Non-habitable planet/moon with an atmosphere
D	Gas Giant
E	Asteroid/small moon
F	Comet

The only thing that changes with the commands we've already discussed, is that you can't put rings on a comet. Speaking of commands, I'd best tell you how to undo any mistakes. Should we add something undesired, then we can get rid of it by…

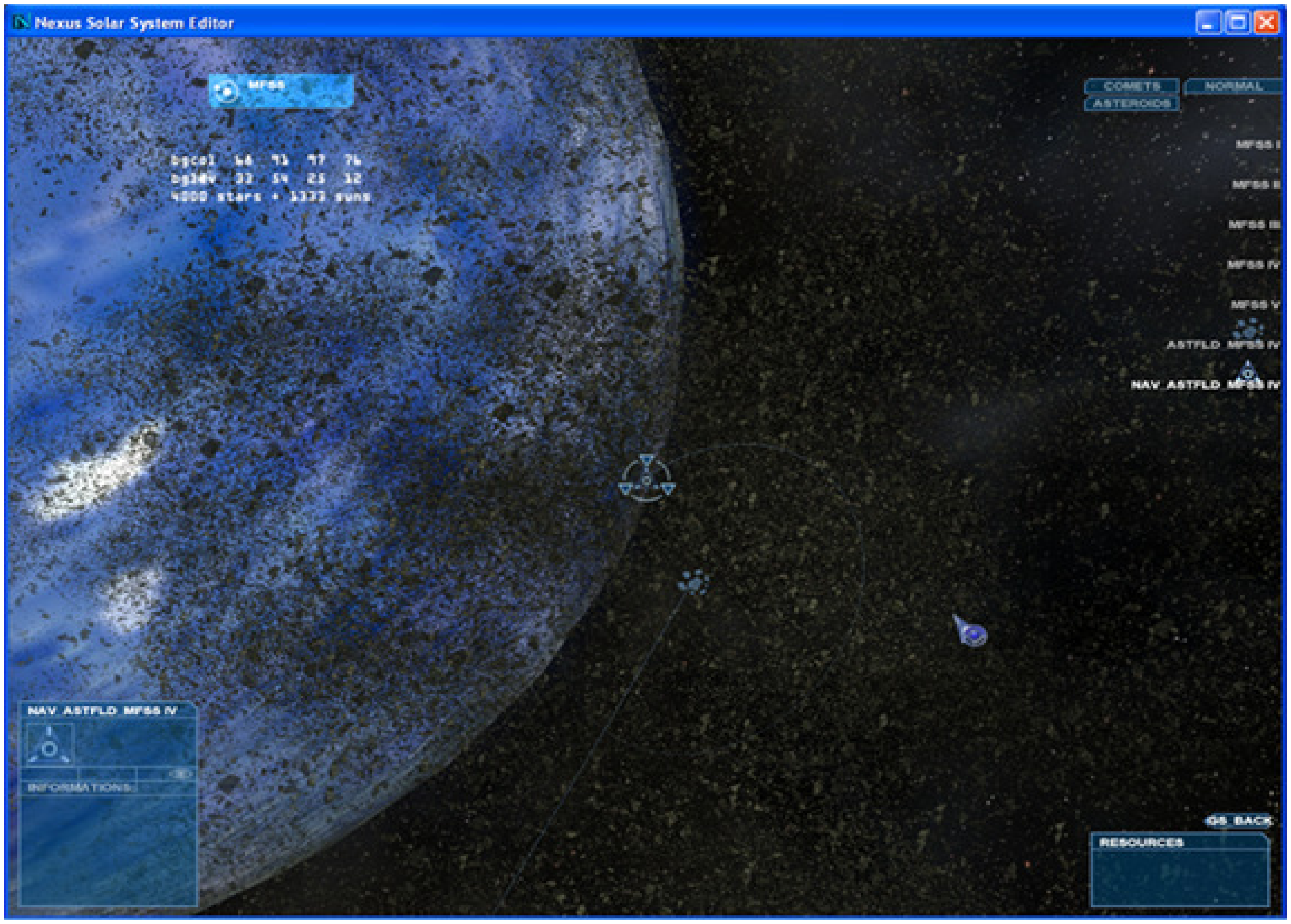
1) Selecting it.
2) Then pressing **CTRL** + **SHIFT** + **DEL**.

And I'd also recommend saving your work at regular intervals (like now for example) by pressing **CTRL** + **S**. You can do this at anytime.

So, now that you're armed with all this information, I'd like you to go off and create a full solar system of planets, moons and what not. Once you're done, come back to me and I'll talk you through adding a few whistles and bells, along with possibly the most important thing we need to add to the system.

All that's left for us to add if we so desire, is an asteroid field or two (Introduced by pressing **CRTL** + **SHIFT** + **I**) or a comet (which, as you can see earlier on this page, is added with **CTRL**+**SHIFT**+**F**). So if you want to add a couple of those, then by all means do so. They obey the same positioning commands as the planetoid (or 'celestial' as the Nexus Devs like to call them) that we first learned on, so you shouldn't have any problems.

Once you're all done, we'll add the final component (or components if you like, as we're not restricted to a single one), which is a **Nav-marker**. As with everything else we've placed so far, barring the sun, select an object then press **CRTL** +**SHIFT** + **N** to add our little marker. Here's mine, should make a nice mission area, don't you think?



```
bgcol  66  91  97  76
bg16v  33  54  25  12
4000 stars + 1333 suns
```

As you can see, I've placed the nav-marker (currently selected) amongst an asteroid field in high orbit over a planet. There's quite a lot you can do with this editor as long as you have the time, patience and imagination to plan it. So once again, take a little time to experiment, then when you're ready, save it one last time and exit the editor with the proper key press of **CRTL** + **Q**.

And that's it; you've not only created your very own solar system, but also completed the third part of this tutorial! But before you go anywhere, if you now navigate your way back to our **systems** folder, you'll now discover a new file in there called 'Mfss.system.bg' (assuming you've followed this tutorial religiously that is). This file is that nice black background filled with stars that you took little notice of while putting your system together – do <u>not</u> delete it, whatever you do or bad things may start happening. You have been warned (imagine the last bit in a deep, booming, ominously godlike voice for a more dramatic effect).

Also, if we open the **Mfss.system** file, you'll also see a lot more data in here. All added automatically by the editor for us so we don't have to wear our keyboards out. Earlier on, I mentioned that you need to save your text files before starting up either editor. Well the same is also true in reverse. If you've had the editor out and done some work but left its corresponding text file open, then close and reopen it before you do anything else (if it asks you to save, you're boned and will have to chose which bit of work to lose) or you'll lose the work.

There are other things you can do with the solar system editor, but I'll leave you to discover those for yourself. Or maybe one day (if I'm feeling brave enough), I'll try and write an 'advanced' tutorial covering whatever else I've managed to figure out by then. Also for your reference, you'll discover a list of the solar system editor's main commands somewhere near the end of this tutorial so you don't have to spend ages searching back and fore as you try to find out how to make planets bigger.

But we’re done with this section now (I know I said that before and then rambled on for ages, but we are now moving forward), so onto the final part… mission scripting – the part that’s going to give you no end of stress, grey hairs and sleepless nights. Welcome to my world!

****

Part Four… Mission Scripting!!

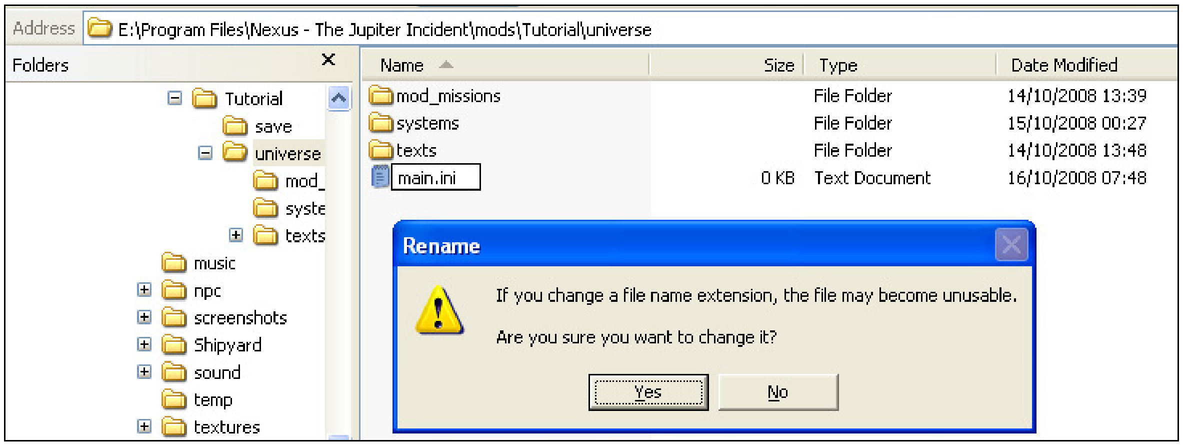
And here we are, in the final section of this tutorial. This is where we'll learn how to place objects within a mission area and then give them instructions on what to do and how to behave. Now before I begin, I'd just like to offer thanks once again to **Jusas** for writing the original tutorial (Missionmaking(for dummies)) which I'm once more using as a foundation. We won't go too deep into the Nexus depths or I'll never finish this tutorial, simply because there's too much (and I sure as heck don't understand half of it!). But by the time you finish this section, you will have a playable mission and know enough to go paddling in the great Nexus ocean. Now, just like the previous section, we can't just open the editor and make missions. It doesn't work that way, we've got a few things that we're going to have to prepare first…

Before we can do anything, we have to decide what's going to happen, what it is we're going to create… a plan. It's all fine and well saying "let's write a mod", but about what? We have to figure this out before we can write it. So, as this is my tutorial, I've decided that the scenario is going to be…

As part of its patrol, a small human fleet is to investigate an asteroid field. After all, there's no telling what could be hiding within from long range sensors – a look see is required. Our ships will 'jump in' and proceed to the field, where upon they will discover some Gorg ships that shouldn't be there. As per convention, we have to warn them and give them a chance to withdraw. After that, we get to kick ass. We'll also have two objectives to achieve as well. The first, is our primary objective and it is to eliminate any hostile presence. The second will, strangely enough, be a secondary objective to ensure the survival of our main ship.

Ok, we now have a simple plan of action, so let's get to work…

Navigate to the **universe** folder you created so long ago and create a new text document. Then rename it to **main.ini**, whenever we change a file type (and we'll be doing it quite often!), our pc will complain that the file might not work, etc. Don't worry, it will, so just say yes…

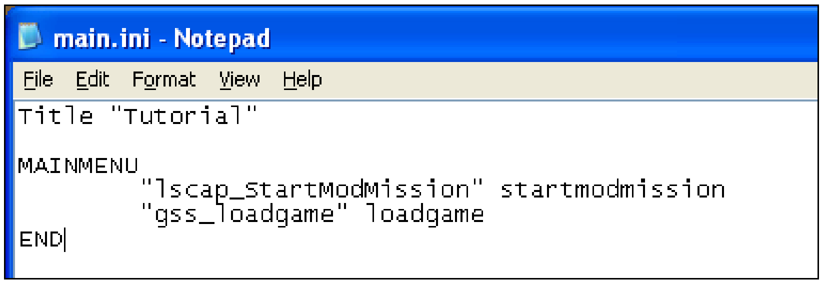


Name	Size	Type	Date Modified
mod_missions		File Folder	14/10/2008 13:39
systems		File Folder	15/10/2008 00:27
texts		File Folder	14/10/2008 13:48
main.ini	0 KB	Text Document	16/10/2008 07:48

If you change a file name extension, the file may become unusable.

Are you sure you want to change it?

Then open our new file and type in the following…

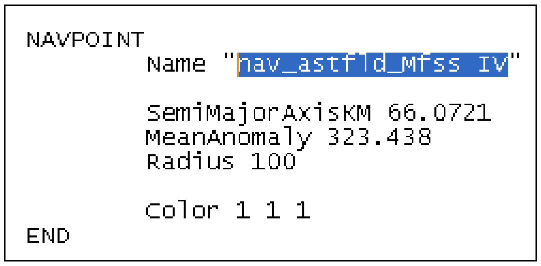


```
Title "Tutorial"

MAINMENU
	"lscap_StartModMission" startmodmission
	"gss_loadgame" loadgame
END
```

Save and close, that's it for this file. What we've done here is make an entry in the menu for our new mod, enabling us to load it into the editor. There are other lines you could enter, but they're not required by us for this mod.

Next up, I'd like you to go to the '**mod_missions**' folder and create a new text file, naming it '**My_First.mission**'. Now this file is where most of the magic will take place and we're going to tackle it in two stages, so… stage one.

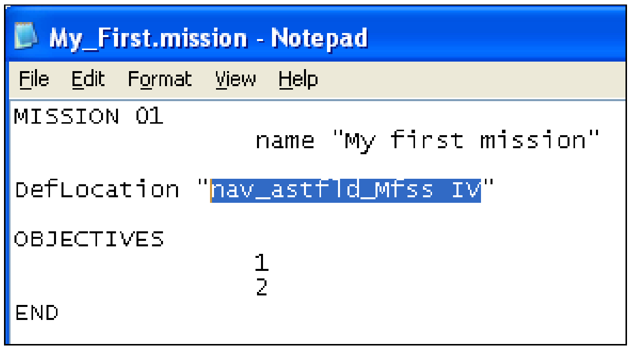
First, make your way to the **solar system** file you created earlier. Open it up and locate a "**NAVPOINT**" entry (if you added more than one, then choose any. It's completely up to you.)



```
NAVPOINT
        Name "nav_astfld_Mfss IV"

        SemiMajorAxisKM 66.0721
        MeanAnomaly 323.438
        Radius 100

        Color 1 1 1
END
```

Select and copy the text in the quotation marks, then close.

Second, go back to the **mod_mission** folder and open the **.mission** file and type the following…



```
MISSION 01
                name "My first mission"

DefLocation "nav_astfld_Mfss IV"

OBJECTIVES
                1
                2
END
```

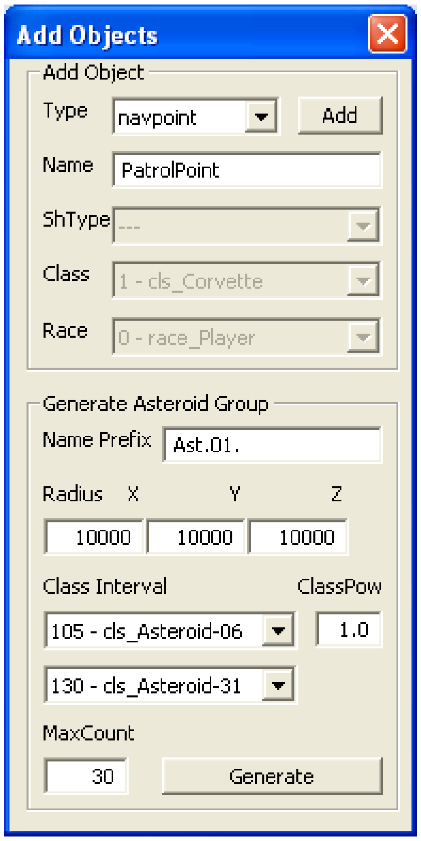
And paste <u>your</u> navpoint name into the highlighted section, save it and close.

Remember I told you that the **nav-marker** was probably the most important thing you could add to a solar system towards the end of part three? Well this is the reason why. Without a '**nav**' location, the game won't know what scenery to load and therefore, won't load our mission. Oh, and while I'm thinking about it. The very first line is important because not only will it be used to denote the missions place in a campaign (should you chose to go down that road) but also, another file that we'll play with later requires it so that it can associate the missions objective with the mission script.

Run the **mod_tools.exe** from the **Nexus\mod_tools** folder once again, select our '**Tutorial**' from the drop down menu and click on the **Mission Editor**. From the menu that appears, we should now be able to select '**My First Mission**' by either double-clicking or high-lighting it, then clicking on the **edit mission** at the bottom of the screen. Do so, and we'll enter our mission area. Just like the **Solar System editor**, this is a "wysiwyg" editor. So everything is graphical at this point. When we start up this beastie, three windows will open, the big window in the middle is where we'll do all our placing and manipulating. A second window (the Nexus Console) will open up behind this main window, you can ignore this window for now as it's mainly used for debugging purposes (just don't try and close it). Finally, the third, smaller window will open up to the left. This is our command pane.

We're not going to go overboard here, all we're going to add is a nav-point, an asteroid field and two small fleets. So, the nav-point first…

1) Click the **Add Objects** button in the command pane. This will open the following window.



Add Objects

Add Object

Type: navpoint — Add

Name: PatrolPoint

ShType: ---

Class: 1 - cls_Corvette

Race: 0 - race_Player

Generate Asteroid Group

Name Prefix: Ast.01.

Radius X: 10000, Y: 10000, Z: 10000

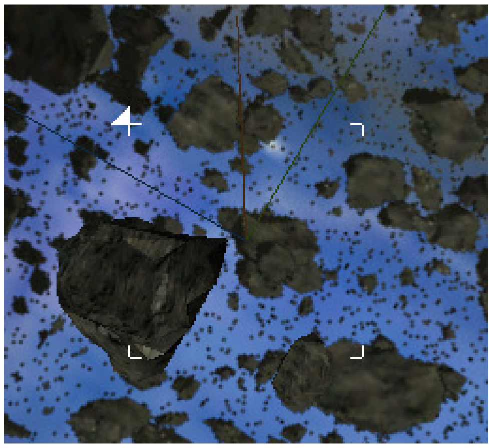
Class Interval: 105 - cls_Asteroid-06, 130 - cls_Asteroid-31

ClassPow: 1.0

MaxCount: 30 — Generate

2) In the drop down **Type** field, select 'navpoint', and rename it to **PatrolPoint** (Just a quick point, you can't enter a 'space' in this field).

3) Click **Add**, and our navpoint will now magically appear in the centre of the scene and be high-lighted. All that marks a navpoint in the editor is a small white arrow when it's selected or the four corner squares when the cursor hangs over it. Like this…



So don't be alarmed if it suddenly disappears, it's still there. Believe it or not, that's as good a place as any for it, so we'll leave that alone and move onto the asteroid field. Once again, click on the **Add Objects** button to bring up the secondary window. Only this time, we're more interested in the bottom half.

Name prefix - As asteroid fields are scenery and very rarely will we do anything but hide amongst them, there's no reason to assign our own name. Best to let the editor handle it.

Radius – Now this is what we want to adjust, we can make any size field we want, just bear in mind that it will always have 'spherical' nature to it (we can stretch it as much as we want, but we can't make weird shapes out of a single field) and the unit measurements are in metres. But for our purposes, we just want a small field, so set all three values to 5000.

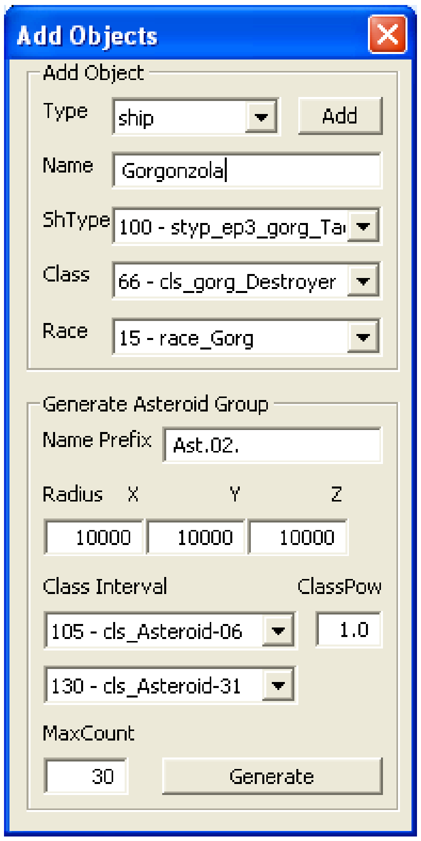
Class Interval – There are actually thirty-five different asteroids we can choose from, with these two fields, we can specify just how many of them we want to use but I normally find the default values to be ok (and I've no idea what the **ClassPow** is).

MaxCount – Self explanatory here, how many 'roids do you want? Just remember, the more you have, the harder your pc has to work! As I said earlier, we only want a small field, so set this value to 100.

And click on **Generate**. Just like the navpoint, this field is just fine where it is.

Just like using the other editor, I'd recommend saving at regular intervals.

Next up, we'll add two Gorg destroyers. As I don't particularly like cheese, we're gonna call them '**Gorgonzola**' and '**Edam**'. So by now you should have already brought up the secondary window. In the **Type** window, make sure that ship is selected. Then, in **ShType**, scroll down till you find number **100**. As you just saw, once you selected it, the **Class** and **Race** fields sorted themselves out… nice, huh? Now all you have to do is fill in the **Name** field and press **Add** and it's in our scene.


Add Objects

Add Object

Type: ship

Name: Gorgonzola

ShType: 100 - styp_ep3_gorg_Ta

Class: 66 - cls_gorg_Destroyer

Race: 15 - race_Gorg

Generate Asteroid Group

Name Prefix: Ast.02.

Radius X: 10000 Y: 10000 Z: 10000

Class Interval: 105 - cls_Asteroid-06, 130 - cls_Asteroid-31

ClassPow: 1.0

MaxCount: 30

Ok, open it up once more and you'll notice that it's still filled in apart from the name, kinda makes life easier when you're bringing in multiple ships of the same type. So change the name ('**Edam**') and add the second ship.

So now that you know how, I want you to add three more ships – a human cruiser (**ShType 135**) named '**Defiance**', along with two destroyers (**ShType 133**) named '**Striker**' and '**Lightning**'. Now that you've added them, it's time to learn about positioning (after all, we can't leave them on top of each other!).

First, let's get the two Gorgs sorted out. Select one by simply clicking on it (or on its name in the upper panel of the Control pane). Also, if you double click on either of these, then you'll zoom in for a close up of the object. Next, hold down **SHIFT** and then move your mouse around… and off she goes! With **SHIFT** pressed we gain access to the abilities of two dimensional movement, so we can now go up, down, left and right to our hearts content. But space is three-dimensional I hear you cry (Well, maybe not you guys, but there are enough voices in my head to make up for you), well, in order to gain access to this third plane of movement, we press **SHIFT** + **LMB** then move the mouse back and fore. So move those two ships to a starting position of your choice (but not too far away, keep them within two thousand meters of our navpoint!).

Next up select all three player ships (**CTRL** + **LMB** and don't worry if they're still on top of each other, we'll sort that in a moment) and move them about 15 k apart (Hint: If you want to know how far apart two objects are, then select one and move the cursor over the second. In the command panes lower window there's a line that tells you). With that done, zoom in on the cruiser (remember, double click on it), then rotate the camera to look at the Gorg ships.

Using the **CTRL** + **LMB** combo, select the cruiser, then the two destroyers (in this order, it's important for the next bit!). Next, by pressing and holding **R** then moving the mouse around, you can rotate the ships as much as you want. What I'd like you to do now, is point our ships towards the navpoint. After all, that's where we're heading.

The final thing we're going to do with these ships, is use a nifty little feature of the editor that'll put our ships in formation. In the previous paragraph, I said to select the cruiser first, this is because the command we're about to use makes the first ship we select the formation leader. So without further ado (and the three ships selected) press **CTRL** + **F**.

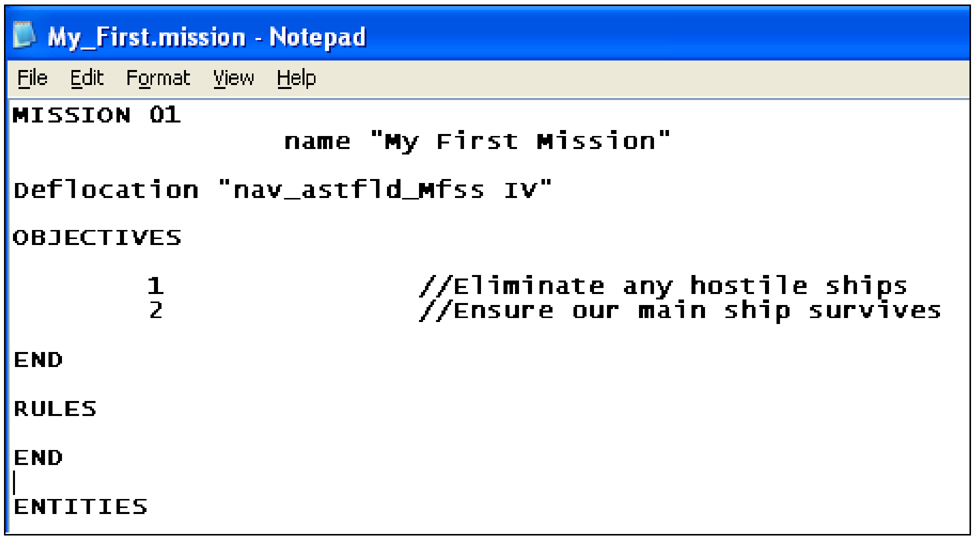
The final thing we have to do now is orientate the two Gorg ships to face our formation. With that done, save the mission (**CRTL** + **S**) and exit (**CTRL** + **Q**). That's the last we'll see of the mission editor now (well, for this mission at least) as everything else that's left will be done with our text editor.

Make your way back to our **mod_missions** folder and open the **.mission** file there, then we'll get 'stage 2' underway. Just like the **solar system editor**, the **mission editor** has filled in an awful lot of 'positional' and 'type' data for us, saving us from a hell of a lot of repetitive typing. Also, before we do delve into this, I'd just like to make you aware of one of my habits that you may wish to take up – but it isn't required by the game in any way.

Throughout the **.mission** text, you'll see the following symbol '**//**'. For those who've scripted before or know a little about the C++ language (on which this is loosely based, I believe), you'll know what it means. But for those that don't, this is known as "commenting out" and basically means we're adding a comment for the human reader to understand and anything behind those to slashes will be ignored by the pc. So I use this feature to write lots of notes about what I'm scripting so when it goes wrong (and it invariably does!) then I can find the relative section without too much brainache.

And <u>save</u> your files often, I cannot stress this enough! (Although you will be after watching a few hours' worth of work disappear at the press of the wrong key!)

So now that you've opened the **.mission** file (and found an **ENTITIES** section has been added), I'd like you to make a few little amendments to the bit you typed before. So, aside from the **Deflocation** line, you should now have…



```
MISSION 01
                name "My First Mission"

Deflocation "nav_astfld_Mfss IV"

OBJECTIVES

        1               //Eliminate any hostile ships
        2               //Ensure our main ship survives

END

RULES

END

ENTITIES
```

Now everything else that we type into this file will go between the two words '**RULES**' and '**END**' and the reason that my demo section is in bold is to make it stand out more for your benefit, not for any technical reason. So let's create our first section…

This is always the '**SceneInit**' rule and as its name suggest, this is where we set most of the names and variables for the mission. From here we'll also be calling the mission **machines** (remember Arparso's bit at the distant start of this tutorial?) that the game will require. So let's fill in what we can of it (because we'll probably becoming back to it time and again to add little bits as we go along) and I'll explain the different bits along the way.

```
RULES

        RULE    event SceneInit

                :action

                        //First of all, let's give all the ships names...

                        M.DE:=GetSceneObj("Defiance");
                        M.ST:=GetSceneObj("Striker");
                        M.LI:=GetSceneObj("Lightning");
                        M.ED:=GetSceneObj("Edam");
                        M.GO:=GetSceneObj("Gorgonzola");

                        //And also the navpoint we placed in the mission.

                        M.PP:=GetSceneObj("Patrol Point");

                :end            //end action

        END             //end rule

END
```

Ok, so we've now inserted our first actual, complete rule into the script. In order for us to issue commands, set conditions or do anything really, all the objects that we plan on doing things with need to be given names… or '**variables**' to be technical. Without these, the game is effectively shouting at everything and nothing is paying the slightest bit of attention, therefore nothing appears to happen when we run our mission (much, much later). You may also notice that I put a space in "**Patrol Point**"? For some reason, we can't put it in with the **Mission Editor**, but we can put spaces in at this point with no problems whatsoever (just don't forget to adjust the name in the **Entities** section too or the game will get confused). And final point before I move on to the next few lines of script, the reason I've put comments on the **END** lines is because you can get a string of up to five **END**'s in a row. As I'm sure you can imagine, this could get awfully confusing! So this is just a measure to help keep things clear.

In this next section we'll add some rules to hide our ships, set our objectives up and make sure the Gorgs are hostile. So add these lines to the **SceneInit**…

```
    //Now we'll select and hide our ships.

    SelectShips(1,S.race=#race_Player);
    ExecList(1,
            Disappear(S.this);
    );

    //Set up the relations...

    SetRelation(#race_Player,#race_gorg,2);
    SetRelation(#race_gorg,#race_Player,2);

    //And now our objectives.

    SetObjectivePrimary(1,1);
    SetObjectivePrimary(2,0);

    MissionObjective(1,#OBJECTIVE_UNCOMPLETED);
    MissionObjective(2,#OBJECTIVE_UNCOMPLETED);
```

You can more or less tell what's going on the comments that I now add by habit, ok so I end up typing more than is strictly necessary but overall, it's easier to figure out what's going on at a glance.

By using the **SelectShips** and then the **ExecList** commands, we can issue orders to a number of objects as opposed to doing them all individually (Oh, you notice I use the phrase '**S.this**' with the **Disappear** command? You'll see it time and again as it generally refers to 'the currently selected objects').

Now the **SetRelation** command only works one way, hence the reason we have two lines that start with the same command. The bit that determines how the two races treat each other is the number at the end, although there are several values that can go in here all we're interested in at this early stage is **1**=**friendly** and **2**=**enemy**.

And finally, the third thing we did is actually split into two sections. The first section determines whether the objective is a primary or secondary objective. The first number in the command selecting the objective number while the second determines its importance (**1**=**primary**, **0**=**secondary**. Basically, it's an on/off switch). The second section decides at what state the objective is at. We can have either **COMPLETED**, **UNCOMPLETED**, **HIDDEN** or **FAILED** as our status.

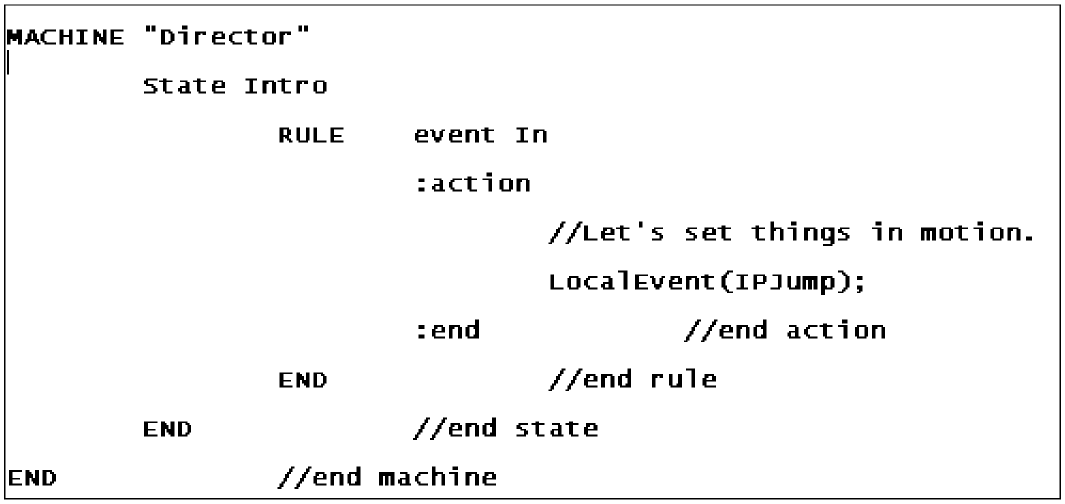
Next…

```
//Let's change from the default music.

PlayMusic(#music_Contact_Gorg);

//Hide the interface for the time being.

GuiSelect(0);

//Set up our initial camera position.

CamLocate(M.DE,2,0,50,2000,M.DE,2,0,0,0,0,0);

//Add the mission title.

Delay(0,1,Title("2107.05.21","My First Mission"),0);

//And here we summon the scenario's machines.

M:GetMachine("Director"):ChangeState(Intro,0);
```

There is a selection of music we can choose from depending on our scenario/mood, just have a look in the **music** folder for what's available (You can also add your own music files too, but that's a bit too advanced for now).

The graphical user interface (or GUI), that nice little bit that enables to us to play the game has a few settings available. In the command we've just issued, we set the GUI to **0**, which removes it entirely. In a little while, we'll return it by resetting the value back to **3**.We've also put our camera somewhere in front of the cruiser and looking at it with the command **CamLocate**. The first half of the command decides roughly where the camera will be and by using the second half, we choose just where to point it.

We're almost done with this section now, just a couple of things left. We're going to put the title up with a slight (one second) delay, just to allow the mission to load in and start up before it shows. The final bit we should always find at the end of the **SceneInit** is the commands to call the machines that we'll be needing. As you can see, the first half calls the actual **machine** while the second half selects what **state** we should be in. So let's take a look at our first machine…



```
MACHINE "Director"

        State Intro

                RULE    event In

                        :action

                                //Let's set things in motion.

                                LocalEvent(IPJump);

                        :end            //end action

                END             //end rule

        END             //end state

END             //end machine
```

Ok, so now we get a peek at what one of these beasties looks like. If you wanted to, you could start your lines tight against the margin or build them across the page as in my example. The reason I've done it this way is because it actually makes it easier to read and spot mistakes (it's all too easy to lose track of which '**END**' you're on). Also while we're here, there are some fixed rule names that we can use in our scripts and for the most part, I'll direct you to the modding manual (our most holy bible!) for these. But there's one that I want to make you aware of, and that's the **RULE** in my example. I didn't choose the name '**In**' at random, the first rule of any state should be named this (even if the rule remains empty!) because as its name implies, this will be the first rule that is to be read. There is also a rule named '**Out**' which, as you'd expect, is the last rule in the state to be read before exiting said state (just for the perfectionists amongst you out there).

Our rule '**In**' only contains one command, a simple summons for the next rule to be executed. So let's slot it in just before the 'end state' **END**…

```
RULE     event IPJUMP
         :action
                  //Enable the camera to get a little closer.
                  CamGoRound(0);
                  //Now we'll create the jumpvector.
                  JumpVector:=VNeg(R2Vec(M.DE:orientation));
                  //Now we'll set the jump direction.
                  SetLongRangeDir(M.DE,JumpVector,0,0);
                  SetLongRangeDir(M.ST,JumpVector,0,0);
                  SetLongRangeDIr(M.LI,JumpVector,0,0);
                  //And now we'll actually call the ships.
                  LongrangeArrive(M.DE);
                  LongrangeArrive(M.ST);
                  LongrangeArrive(M.LI);
                  //And finally, we'll use a timer to call the next rule.
                  Timer(Dialog,7,0);
         :end              //end action
END               //end rule
```

Ok, so what have we done here? Well, you've got my really short explanations within the script, so you could skip on down to the next section if you really wanted to. But for those polite folks determined to stick with me no matter how much I ramble, I'll endeavor to give a few more details.

First of all, we told the camera not to avoid anything (**CamGoRound**), meaning it'll plow straight through everything in its path so you could end up inside an object if you're not carefull with this command. Then we went on to the first and second required parts of getting ships to make a dramatic entrance. Now I'll be totally honest, I don't know what a Vneg is, much less an R2Vec. So I'll have to assume that the **JumpVector** line used in conjunction with the **SetLongRangeDir** lines is us basically saying to the pc "take a look at this ship (in this case, the Defiance), 'cause that's the direction we want to be travelling in". I might be right, I might be wrong, but it's a good enough delusion for me. So let's move onto the third required segment, actually calling the things. Hmm… that kind of sums it up really. Damn.

Ok, the final command we've issued is for the next event, as you've probably figured out already, but we're calling it with a **Timer**, not a **LocalEvent** command. The reason here being that we can't have our captain giving orders before he's actually arrived on the scene, now can we?

So let's get that rule in place…

```
RULE    event Dialog
        :action
                //Show the navpoint.
                MakeFullyKnown(M.PP);
                //Our captain says...
                Delay(0,4,Dialog("PlayerOne",0,0),0);
                //Let's get the camera moving, kinda looks boring just sitting here.
                Timer(CameraRotate,1,0);
        :end            //end action
END             //end rule
```

By now, you know how things work, so let's take a look at the two new commands… In order for us to use a navpoint, we have to make it visible. As it is, the **MakeFullyKnown** command will show the designated object to everything. If we wanted to, by adding a **race** to the command, we could make it visible only to them.

A 'Delayed' **Dialog**. Well, the **delay** part is self explanatory – it's there to stop something from happening straight away and measures the interval in **seconds**. The only thing to remember here, is to add the **variable** section onto the end (Nexus gets quite upset if you don't!)

Which leaves the **Dialog** section. The bit between the brackets is broken down into three sections – the first is the name of the actual dialog file that is to be played (we'll go more into this later), the second part is where you wish this dialog to come from. We could put in the Defiance's variable in this example, but it isn't particularly important (considering what we're going to do at the end of this file) so I've put in a value of '**0**' for this field (again, I'll explain more at a better moment). Which leaves the final part – its priority rating, you can decide if one bit of dialog is more important than another one if you choose to, but I tend to leave this alone personally.

In the next rule that we've called (**CameraRotate**), we'll get our camera to circle our little fleet and reset it to avoid everything. We'll also return the **GUI** so we can play the mission and the final thing we'll cover is how to change **states**.

So the first thing to explore here is the **CamOrient();** command. As per normal, it just does what it says on the can – we're going to change the camera's orientation, which requires five bits of information.

1) We need an object with which to position the camera, so we put in an objects variable (in this case, our cruiser – the Defiance).
2) Our heading in degrees. This is one of two ways, I'm not quite sure which. Either consider the Defiance's prow as 0 degrees or the closest side of the object to you as 0 degrees and work out just where you want the camera.
3) The pitch (again, in degrees). In case you want to go above or below the object as well.
4) The distance in meters we want the camera to end up.
5) The time in seconds. How long we want the camera to take when travelling from the start position to the end position.

```
        RULE    event CameraRotate

                :action

                        //With this little command, we'll move the camera.

                        CamOrient(M.DE,175,20,1000,15);

                        //And reset the controls.

                        Delay(0,12,CamGoRound(2),0);
                        Delay(0,12,GuiSelect(3),0);

                        //Finally, let's change the state.

                        Delay(0,15,ChangeState(Survival,0),0);

                :end            //end action

        END             //end rule

END             //end state
```

Well, you already know about the **CamGoRound** and **GuiSelect** commands, so we'll leave them alone. Which leaves our method of calling another machine state. Just like the **LocalEvent** command, we simply name the state that we want to enter. But unlike the **LocalEvent**, we also have a variable section to fill in. In our example here, we've left it as a simple '**0**', but later on we'll do one with a variable entry. You'll also notice that as we've reached the end of a state, there's an extra **END** involved. Right, on with the next state then…

```
STATE Survival

        RULE    event In

                :action

                        //The first thing to do here, is make sure that the Gorgs are seen.

                        Gorg:=GetFreeSel();
                        SelectShips(Gorg,S.race=#race_Gorg);
                        ExecList(Gorg,
                                MakeFullyKnown(S.this,#race_Player)
                                );

                        //Next we'll add a little more to the 'story'.

                        Delay(0,2,Dialog("PlayerTwo",0,0),0);
                        Delay(0,5,Dialog("GorgOne",0,0),0);
                        Delay(0,10,Dialog("PlayerThree",0,0),0);

                        //Here we'll call the Gorgs AI machine.

                        Delay(0,5,M:GetMachine("GorgAI"):ChangeState(Attack,0),0);

                :end            //end action

        END             //end rule

END             //end state
```

Before we go into this little block, I'd just like to introduce you to another little habit that I've picked up along the way. Whenever I start a new script block (doesn't matter if it's **MACHINE**, **STATE** or **RULE**) I always write all the '**ins**' (you know, the **rule**, the **:action**, etc) and all the '**outs**' (all those '**END**'s). I just find that it helps me avoid forgetting the odd little bits (Hmmm, that'll be the old age and Alzheimer's kicking in, Old Dragon!) here and there. But back to the new state!

Ok, the first thing we did here was to select both the gorg ships and make sure that they're visible to the player (chances are that they already are as we're 15k apart, but we'll just be sure). If you think back a little, we've already talked about most of the first block, the only differences here being that we've added a 'race' variable to the **MakeFullyKnown** command, and that we've put a new command at the start.

The **GetFreeSel();** command is a means of allowing us to create a list with a name of our choosing (as opposed to simply using numbers, which in a complex mission, could see us ending up forgetting what list they actually refer to. Not good). Also, now that we have a named list, we don't need to go through the **selectships** routine again to select the gorgs. We can skip straight to the **ExecList** bit! See, the devs do care about our finger tips. Though I wouldn't rely on the list being transferred from machine to machine, best to make a new copy if you wish to do that.

Oh look, some delayed dialog, we've already been through that so I'd just like to say… if you can, try to figure out the correct 'delay' time for each bit of dialog. It's not vital to the dialog itself (we could change the delays to 2,3 and 4 seconds respectively and they would still come out in the correct order, one after the other), it's just that if you're scripting something else to happen alongside this, the dialog makes for a good 'time' benchmark.

The final thing that we did in this rule was to call the **machine** that will control the enemy ships. We could have put this in the **SceneInit** section, but that would require us to play about with timing issues and also have a **TICK** rule running before we really need it to. This way, things are a little neater.

Next rule…

```
RULE    event Tick
        :action
                //Call the end if the cruiser gets wasted.
                If(M.DE:wreck=1,ChangeState(TheEnd,E.Survival:=0));
                //Also end the mission if the gorgs are taken out.
                If(SelectShips(3,S.race=#race_gorg)=0,ChangeState(TheEnd,E.Survival:=1));
        :end            //end rule
END             //end state
TICK 5
```

And welcome to our very first '**TICK**' rule. The idea behind these rules is to continually check a certain set of conditions at a selected interval (in this case, every five seconds). You see the line at the bottom that says '**TICK 5**'? This is where we set the checking interval. Without this present, the rule will <u>not</u> run, so it is very important. However, you only need to put this line in once per state as all the **TICK** rules within that state will obey it. So let's take a look at the commands within the rule shall we?

With the first command line, we're getting the game to check that the cruiser is still alive. Should it not be, then we'll change to one of two possible endings all thanks to the variable we've set.

In the second line, we're doing exactly the same thing but with the gorg ships. Should this condition become true (i.e. there are no gorg left), then we get taken to the second ending we're going script.

Wow, that's the end of that state, thought I'd have to add more to it. Guess I'd forgotten how simple things can be sometimes…

Now for the final state in this machine.

```
        STATE TheEnd
                RULE    event In
                        :action
                                //First of all, the bad ending.
                                If(E.Survival=0,
                                        MissionObjective(1,#OBJECTIVE_FAILED);
                                        MissionObjective(2,#OBJECTIVE_FAILED);
                                        GuiSelect(0);
                                        Title(0,0,0,"A fiery end to a promising career");
                                        Delay(0,7,FadeScreen(1,3,0),0);
                                        Delay(0,10,uQuitMission(0),0);
                                );
                                //Then the good ending.
                                If(E.Survival=1,
                                        MissionObjective(1,#OBJECTIVE_COMPLETED);
                                        MissionObjective(2,#OBJECTIVE_COMPLETED);
                                        GuiSelect(0);
                                        Title(0,0,0,"Bask in the glory of your victory");
                                        Delay(0,7,FadeScreen(1,3,0),0);
                                        Delay(0,10,uQuitMission(1),0);
                                );
                        :end            //end action
                END             //end rule
        END             //end state
END             //end machine
```

And there we have it in its entirety. As you can see, it's made up of two '**If**' statements - the first to handle failures and the second for success. It's mainly self explanatory and we've actually addressed most of these commands before so there's not much more for me to add really.

As much as I'd like to say "there you are, now go and play your new mission…", I'm afraid it'll crash at the moment due to a command we put in the **Survival** machine – the one where we call another machine. So let's rectify that by making a new machine that'll handle the gorgs engagement rules. Add the following lines to the **.mission** file.

```
MACHINE "GorgAI"

        STATE Attack

                RULE    event In

                        :action

                                //Select the gorgs and issue these orders...

                                Gorg:=GetFreeSel();
                                SelectShips(Gorg,S.race=#race_gorg);
                                ExecList(Gorg,
                                        MakeFullyKnown(M.DE);
                                        S.this:tMoveTo(M.DE,2,0);
                                        S.this:tEnergyMode(1,2);
                                );

                        :end            //end action

                END             //end rule

        END             //end state

END             //end machine
```

Another short and sweet machine. Just to be safe, we've repeated the Gorg list because sometimes the game forgets these things from machine to machine. In laymans terms, what we've done here is write a really basic and dumb AI script which will make the gorgs chase our cruiser around. Just to give them a little bit of bite too, we've overpowered their weapons so they'll be firing at anything that moves.

Ok, we're almost done with this file. Just two more little bits to add now, we're going to add characters to two of the ships. So scroll down through the **ENTITIES** section until you find the Defiance's entry and amend it to read…

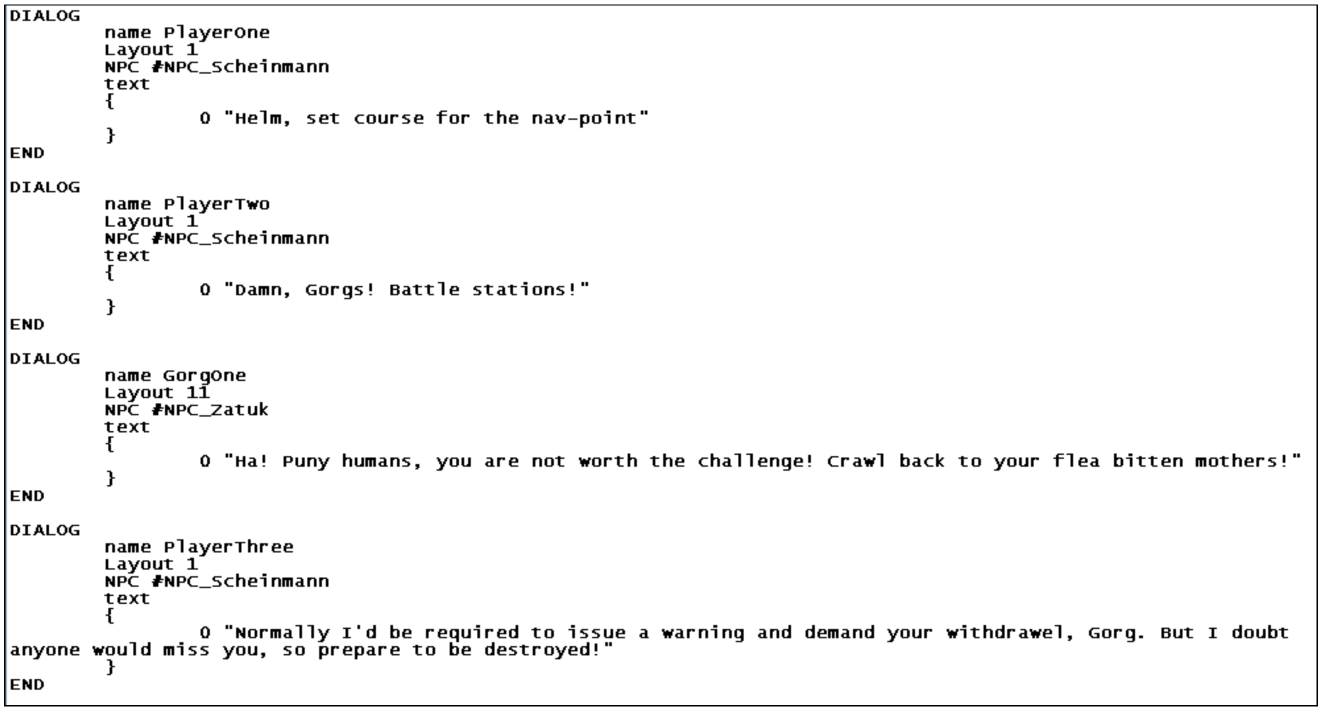
```
SHIP
        Name "Defiance"
        Race #race_Player
        ShipType #styp_ep4_Heavy_Cruiser
        Position -14975.4 6237.79 -3482.82
        Orientation -81.9562 -23.7071 30.5558

        NPC
                Name "Scheinmann"

                ID 10
                Face 28
        END
END
```

And the Gorgonzola's to read…

```
SHIP
        Name "Gorgonzola"
        TechCat 31 20
        TechPointsScanned 0
        Race #race_Gorg
        ShipType #styp_ep3_gorg_Tactical_Destroyer
        Position 773.638 678.982 -701.729
        Orientation 98.6076 17.3807 -56.585

        NPC
                Name "Zatuk"

                ID 69
                Face 50
        END
END
```

And hey presto, you've just written your first mission script, HOORAH!!! (Feel free to run round wherever you are for a victory lap ☺). But before you get too carried away in your celebrations, we've still got two more short files to write before this mod is complete. So now you've got to navigate your way to the **Universe/texts/dialogs** folder and create a new file called '**My_First.ini**'.

Now this file is where we're going to write all the text that is going to appear on the screen and make up our little 'plot'. So open **My_First.ini** and add these lines…



```
DIALOG
        name PlayerOne
        Layout 1
        NPC #NPC_Scheinmann
        text
        {
                0 "Helm, set course for the nav-point"
        }
END

DIALOG
        name PlayerTwo
        Layout 1
        NPC #NPC_Scheinmann
        text
        {
                0 "Damn, Gorgs! Battle stations!"
        }
END

DIALOG
        name GorgOne
        Layout 11
        NPC #NPC_Zatuk
        text
        {
                0 "Ha! Puny humans, you are not worth the challenge! Crawl back to your flea bitten mothers!"
        }
END

DIALOG
        name PlayerThree
        Layout 1
        NPC #NPC_Scheinmann
        text
        {
                0 "Normally I'd be required to issue a warning and demand your withdrawel, Gorg. But I doubt
anyone would miss you, so prepare to be destroyed!"
        }
END
```

As you can see, we've got four separate blocks of code here, the layouts always the same so let's take a look at the first one…

DIALOG This informs our pc to start taking note.

Name What we've decided to call our file, can be anything really. Just remember that it's your fingertips you're wearing out with long file names!

Layout This denotes where on the screen our dialog box will appear. You can find more precise information about it somewhere in the manual.

NPC Well… who's doing the talking, obviously.

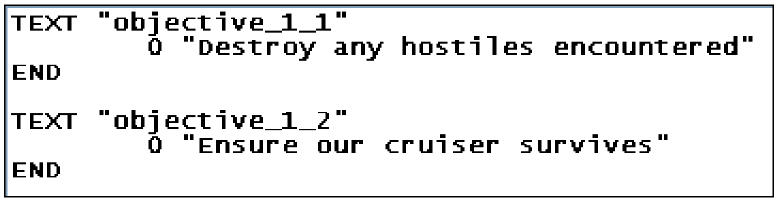
Text Just warns the pc that the actual text's coming up. As far as I can tell, the 0 is pretty important as it tells the game we're speaking English.

END Tells the pc to go back to sleep.

There are a couple of other little bits we could throw in here, but they're not really required for this mod.

Ok, one down… one to go. Now make your way to the **universe/texts/texts** folder and create another file called '**My_First.ini**'. Don't worry, the pc won't get confused (we might, but it won't), it knows the score here.

Open it up and type the following…



```
TEXT "objective_1_1"
        0 "Destroy any hostiles encountered"
END

TEXT "objective_1_2"
        0 "Ensure our cruiser survives"
END
```

This is where we store the text that'll appear when you open up the objectives window during the mission. Without this, all it shows is the 'objective_1_1' bit.

****

Closing Words

And that would be about it folks, you've now written everything you need to have a playable mod! It might not be the best thing since sliced bread, but it's yours and it works (hopefully☺). Hopefully, from here, you'll start experimenting with the various editors and creating your own mods , I'd recommend reading other mod scripts and the manual for further reference and there are several forums that you can visit too should you run into problems (and you will). Beyond this point, aside from possibly more of my rambling, you'll find a few more pages of text containing reference lists for the two editor, a list of web addresses and god knows what else that I add before this thing gets posted.

Knowing what an inquisitive and impatient little bunch of imps you lot are, you've probably already found and jumped straight into it; But I've bundled up the **Tutorial mod** that I wrote while creating this document (well, had to make sure I was giving you working information, didn't I?) for you, so should you find yourself getting lost or confused along the way… this is what it should be looking like.

And finally, just before I sign off, I would like to know your thoughts on this tutorial – the good, the bad and the ugly. Should I end up writing another tutorial for Nexus (or any game for that matter) then it'd be helpful to know just what you folks thought of this one. Keep scrolling down and you'll find a few ways to contact me in one of the appendices.

All the best in your own modding…

Kind regards,

The Old Dragon.

Appendix A – Solar System Editor commands

CRTL + SHIFT + A	Create habitable plant/moon
CRTL + SHIFT + B	Create non-habitable planet/moon with an atmosphere
CRTL + SHIFT + C	Create non-habitable planet/moon without an atmosphere
CRTL + SHIFT + D	Create gas Giant
CRTL + SHIFT + E	Create asteroid/small moon
CRTL + SHIFT + F	Create comet
CRTL + SHIFT + I	Create asteroid field
CRTL + SHIFT + N	Create nav-marker
CRTL + Q	Quit editor
CRTL + S	Save file
CRTL + SHIFT +U	Create sun

You need to have an item selected for these commands to work.

CRTL + SHIFT + DEL	Delete currently selected item
SHIFT + Letter **A** – **E**	Generate a new appearance
R	Turn celestial rings on, cycle through, then turn off
Home, **End**	Adjust outer radius
CTRL + **Home**,**End**	Adjust inner radius
SHIFT + **Home**,**End**	Adjust opacity
Numpad + and **-**	Change size/intensity of sun or planet
Numpad / and *****	Change colour/temperature of sun
Numpad 4 and **6**	Cycle through the planet colours/textures
Numpad 1, **2**, **3**	Increase planet 'glossiness' (R, G, B respectively)
CRTL + **Numpad 1**, **2**, **3**	Decrease planet 'glossiness' (R, G, B respectively)
PG UP, **PG DN**	Adjust the orbital path radius
CRTL + **PG UP**, **PG DN**	Adjust orbital path 'regularity'
Up/**Down** arrow keys	Change the angle of the orbital path
Left/**Right** arrow keys	Move the object along the orbital path

Appendix B – Mission Editor commands

Double-click on an object	Will zoom you in for a close up, moving the mouse around now will cause you to rotate around the object. A single click on another object now will turn the camera to show both objects in the view screen.
CTRL + **F**	Will place all the selected objects into formation with the first object selected as the formation leader.
R	Pressing and holding '**R**' with an object (or objects) will cause that object to rotate in place. Used for orientation purposes. When used on multiple objects, their current places in relation to each other will be preserved.
CTRL + mouse movement	Pressing **CRTL** while moving the mouse around will cause the selected object to move in two dimensions.
CTRL + LMB + mouse movement	Pressing **CRTL + LMB** while moving the mouse around will cause the selected object to move in the third dimension.

Appendix C – Sites of Interest

First of all folks, should you wish to let me know your opinions on this tutorial, you can post on any of the following forums (I tend to haunt them on a regular basis – sad I know) or you can drop me an email directly at…

draconicdreams@hotmail.co.uk

Now those forums…

At the time of this writing, I'm afraid that the official sites and forums of **Nexus:The Jupiter Incident** appear to have been laid to rest (Oh woe is us (doffs cap in respect)).

The SWAT-Portal Forums (I'm afraid you'll have to scroll down to the Nexus section in here).

http://www.swat-portal.com/

The Nexus:War Begins mod forum.

http://www.theimperialassault.com/orderoftheblackknights/index.php?act=idx

There are also a couple of ModDb sites you can visit (the mods that these relate to were in production at the writing of this tutorial), they are…

Nexus:The Aurora Affair

http://www.moddb.com/mods/nexusthe-aurora-affair

Nexus:War Begins

http://www.moddb.com/mods/stargate-mod-war-begins