Creating Custom Systems For PlanetFall2

This document is designed to help anyone to create fully custom systems, regardless of your programming ability.

There are three components to making a fully custom system.

1. **shipdata.plist entries**
   Create any unique landing sites in this file. You don't *need* to do this – it's only required if you want to create a unique location, as distinct from using one of the built-in ones.
2. **Landing images**
   If you want the arrival report to have a custom image for your unique landing site, they can be added to the worldScripts.PlanetFall2._landingImages dictionary. Once again, these aren't necessary, but if you're creating a unique location, then having some eye-candy for the player can be rewarding.
3. **System definition**
   Define what sites you want to have on the main planet, give each location a name, and additionally define sites for any extra planets and moons as well.

OXP package structure

I said this was going to be for everyone! If you already know how to create the structure for an OXP, feel free to skip ahead. But for everyone else, read on.

An OXP is made up of files and folders, and those files and folders need to be in specific locations. First, you need the base folder for your OXP. For the purposes of this document, our OXP name is going to be "SuperSystems". So, in the AddOns folder of your Oolite installation, you need to create a folder called "SuperSystems.oxp"
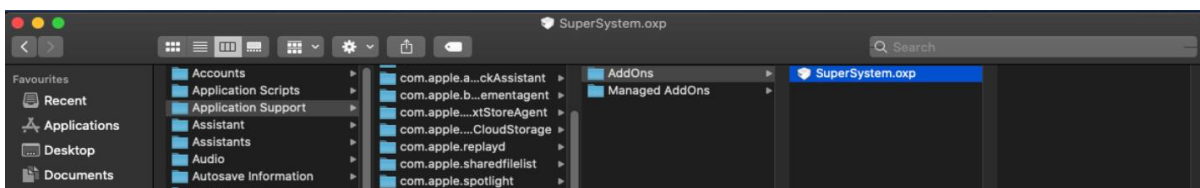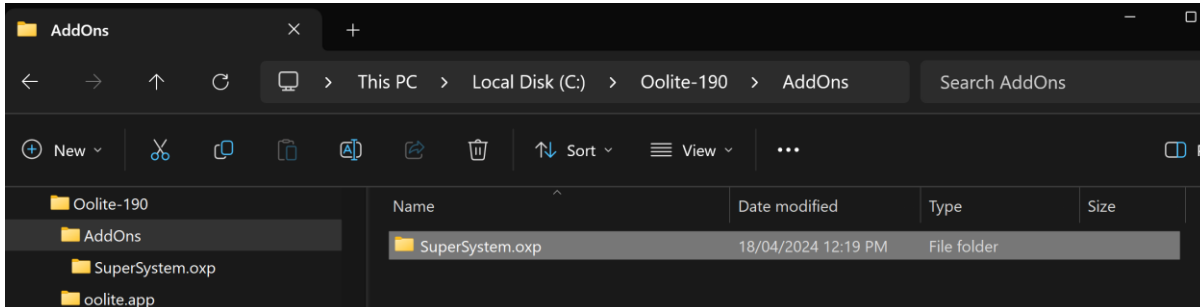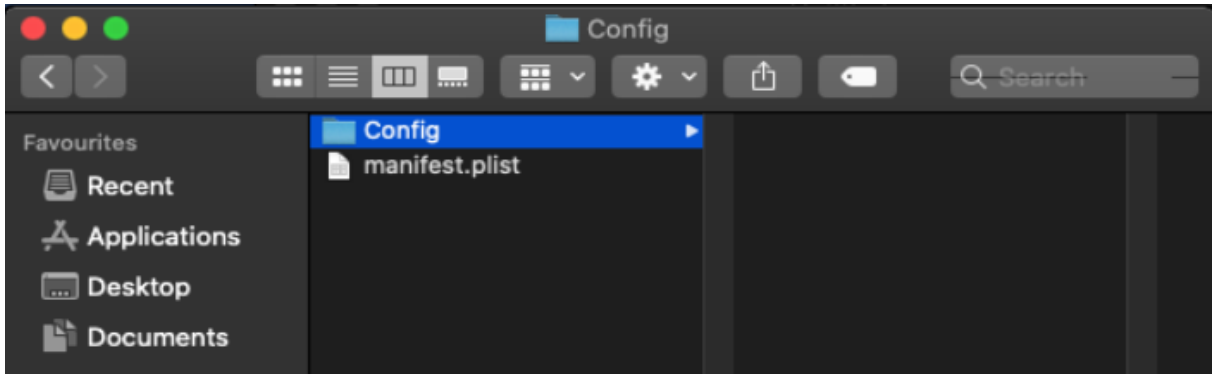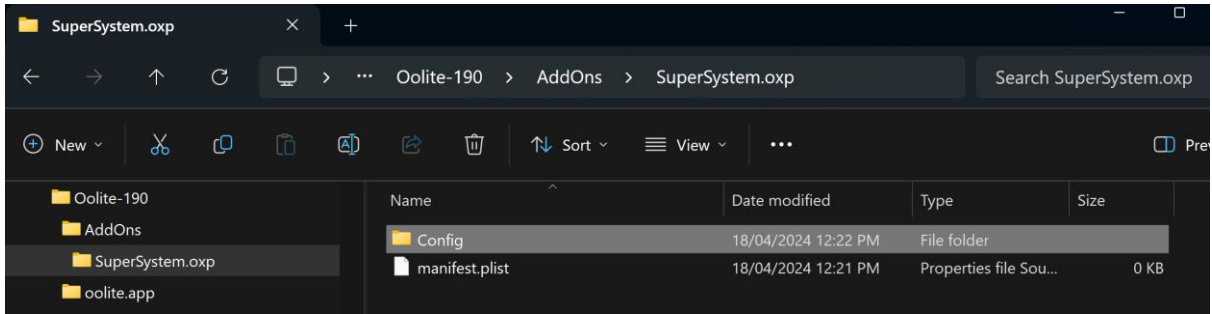

Then inside this new folder, we're going to create some other things.

First, we create a "manifest.plist" file. This file is a plain text file, which we'll write up later.
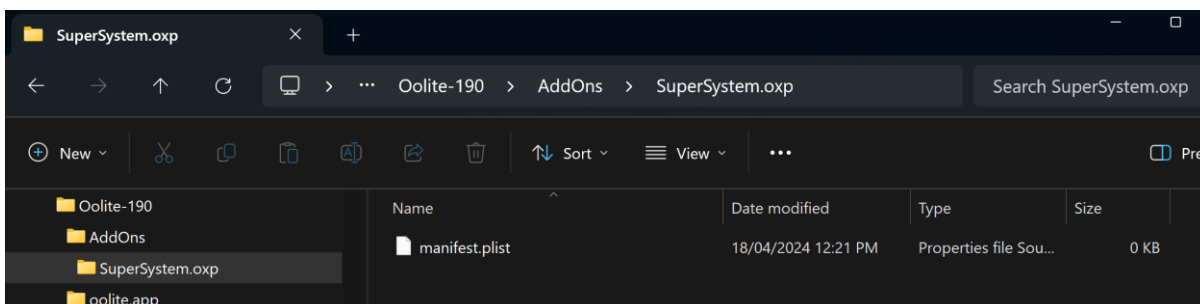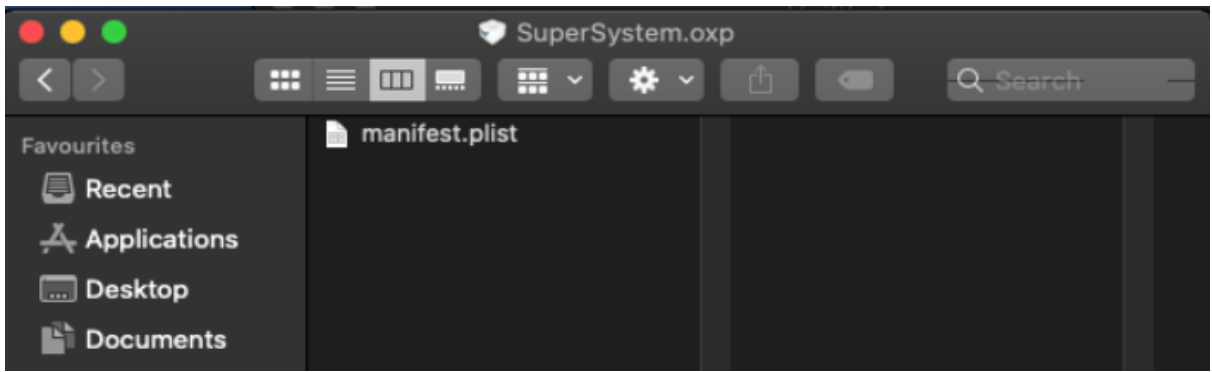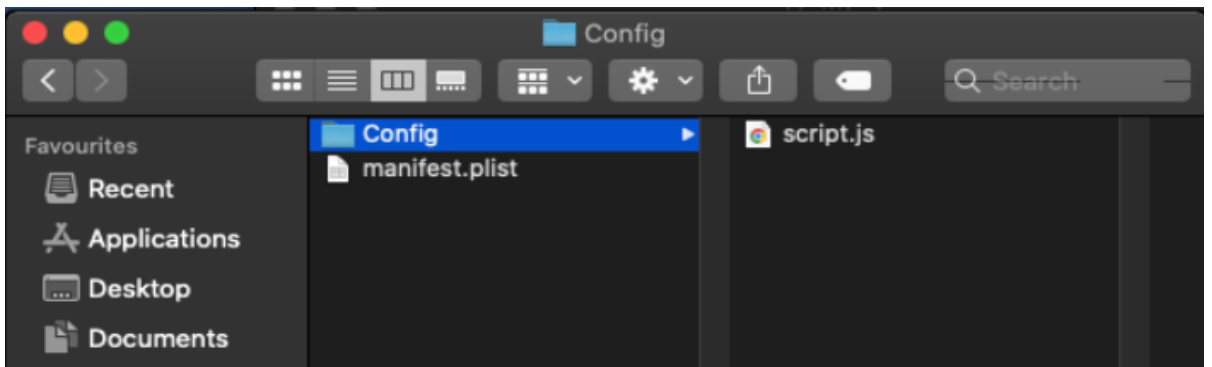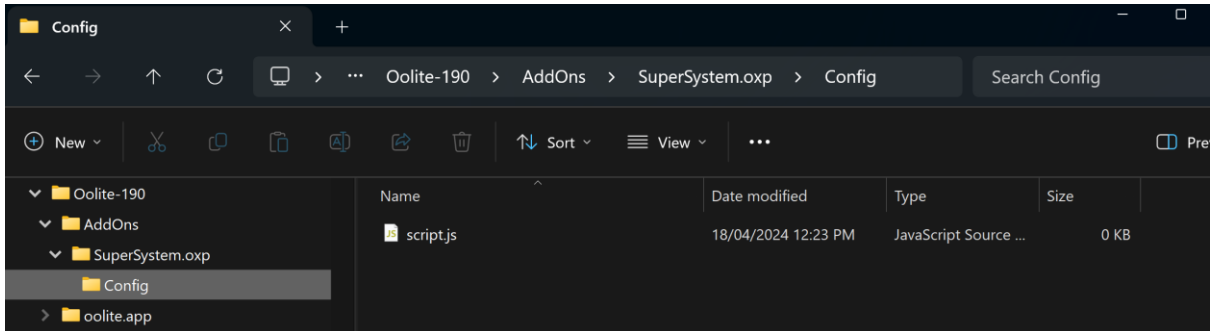
Next, we need to create a "Config" folder.

Finally, inside the Config folder, we're going to create a file called "script.js"

That's all we need to get started. Now, let's go back to that manifest.plist file and put some data in. Edit the file using a standard text editor. That application will be different, depending on your platform (Windows, Linux or

Mac). Don't use a word processor (eg Microsoft Word, LibreOffice Writer, etc), as those applications can insert control characters into your files that might confuse Oolite.
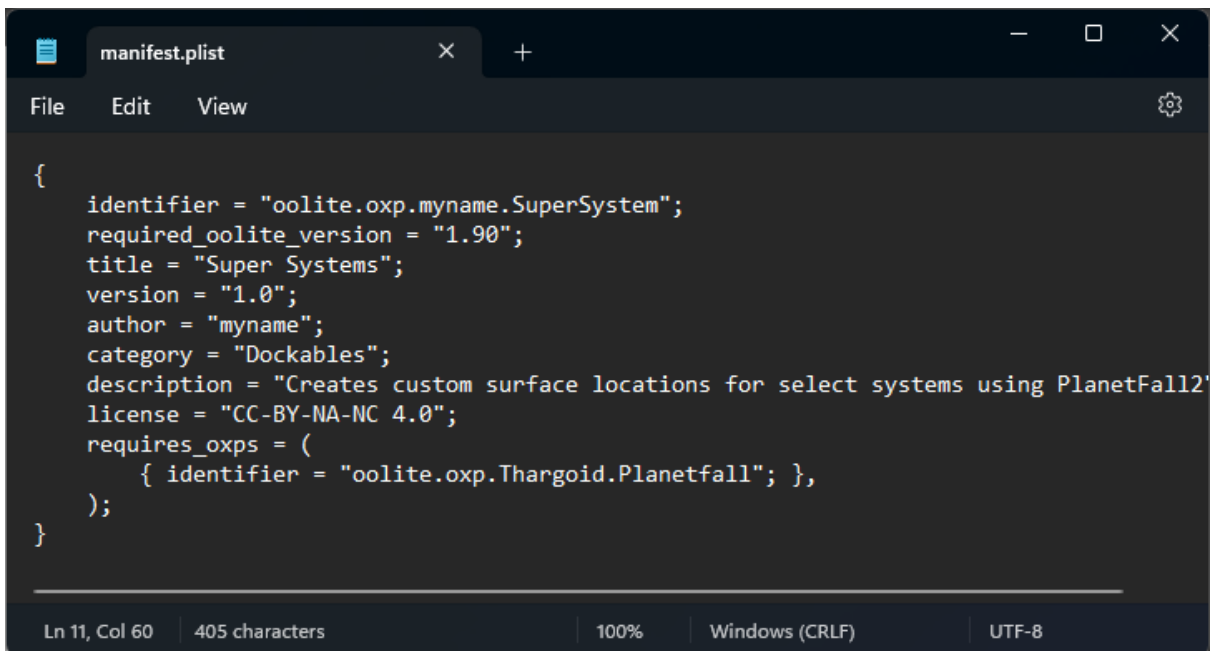
Here I'm using Notepad that comes with Windows 11 (note, for Windows version 10 and previous, some users have found issues saving documents in the correct format. The Windows 11 version seems to be free of these issues). I've also included screenshots from Mac using TextEdit.
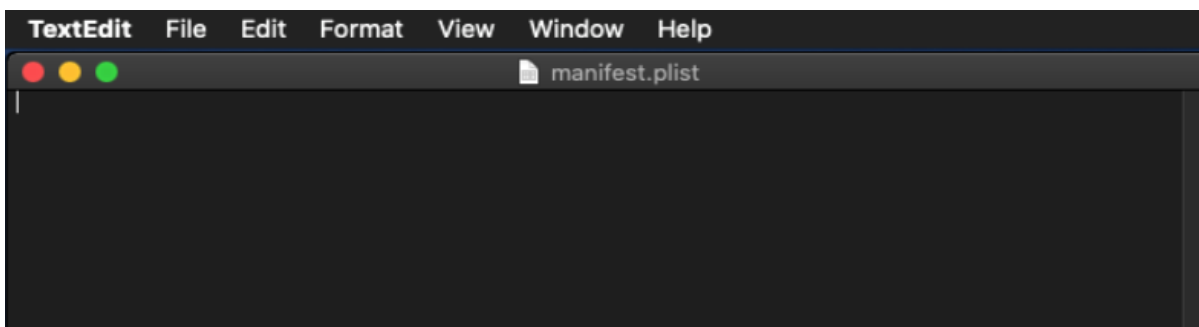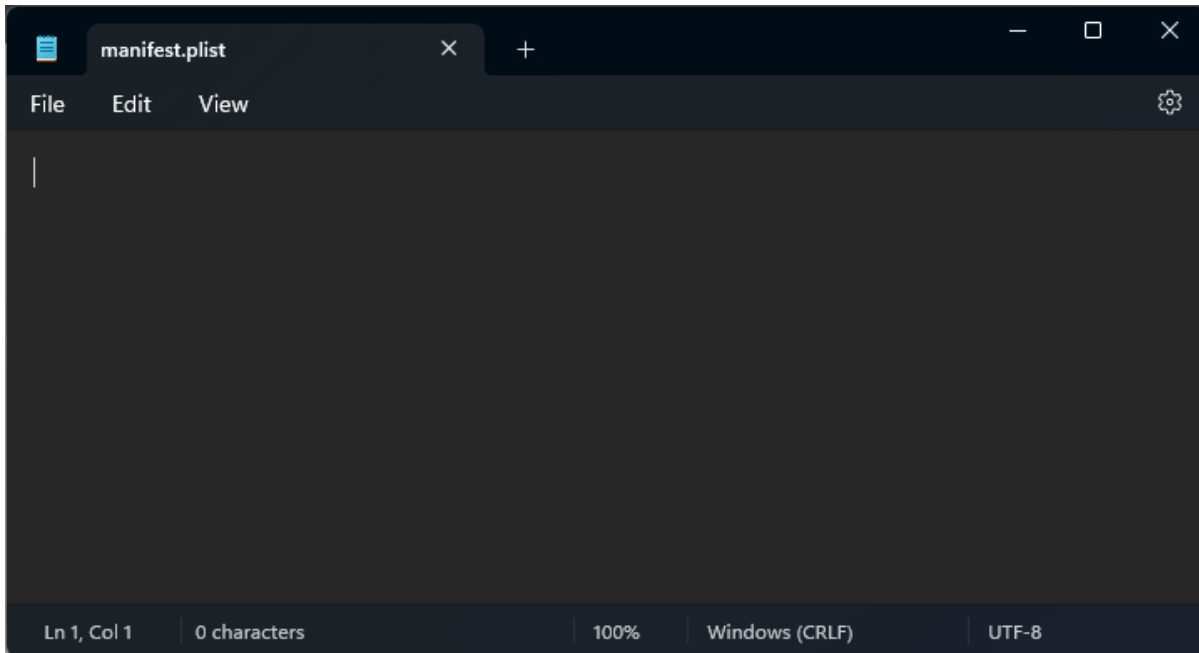
The minimum amount of information you need to enter here looks like this:

```
{
    identifier = "oolite.oxp.myname.SuperSystem";
    required_oolite_version = "1.90";
    title = "Super Systems";
    version = "1.0";
    author = "myname";
    category = "Dockables";
    description = "Creates custom surface locations for select systems using PlanetFall2";
    license = "CC-BY-NA-NC 4.0";
    requires_oxps = (
        { identifier = "oolite.oxp.Thargoid.Planetfall"; },
    );
}
```

It should look something like this:

```
{
    identifier = "oolite.oxp.myname.SuperSystem";
    required_oolite_version = "1.90";
    title = "Super Systems";
    version = "1.0";
    author = "myname";
    category = "Dockables";
    description = "Creates custom surface locations for select systems using PlanetFall2"
    license = "CC-BY-NA-NC 4.0";
    requires_oxps = (
        { identifier = "oolite.oxp.Thargoid.Planetfall"; },
    );
}
```

Ln 11, Col 60   405 characters   100%   Windows (CRLF)   UTF-8

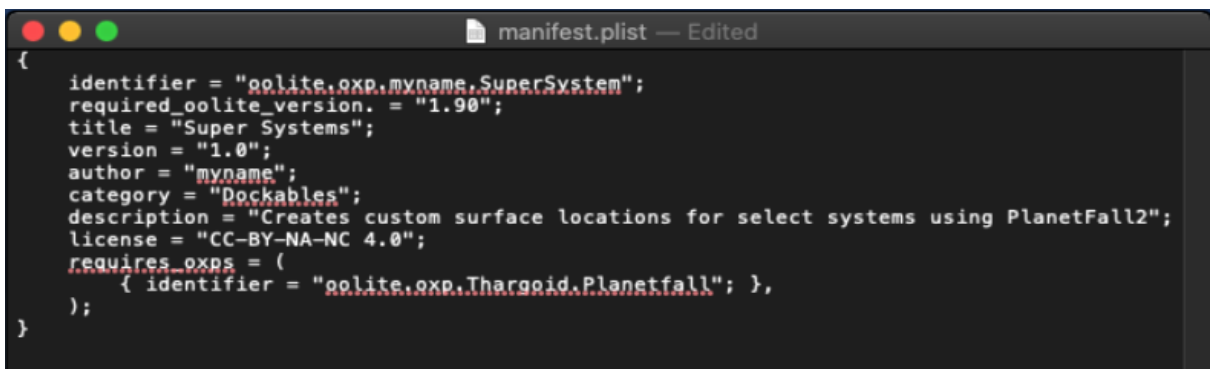manifest.plist — Edited

```
{
    identifier = "oolite.oxp.myname.SuperSystem";
    required_oolite_version. = "1.90";
    title = "Super Systems";
    version = "1.0";
    author = "myname";
    category = "Dockables";
    description = "Creates custom surface locations for select systems using PlanetFall2";
    license = "CC-BY-NA-NC 4.0";
    requires_oxps = (
        { identifier = "oolite.oxp.Thargoid.Planetfall"; },
    );
}
```
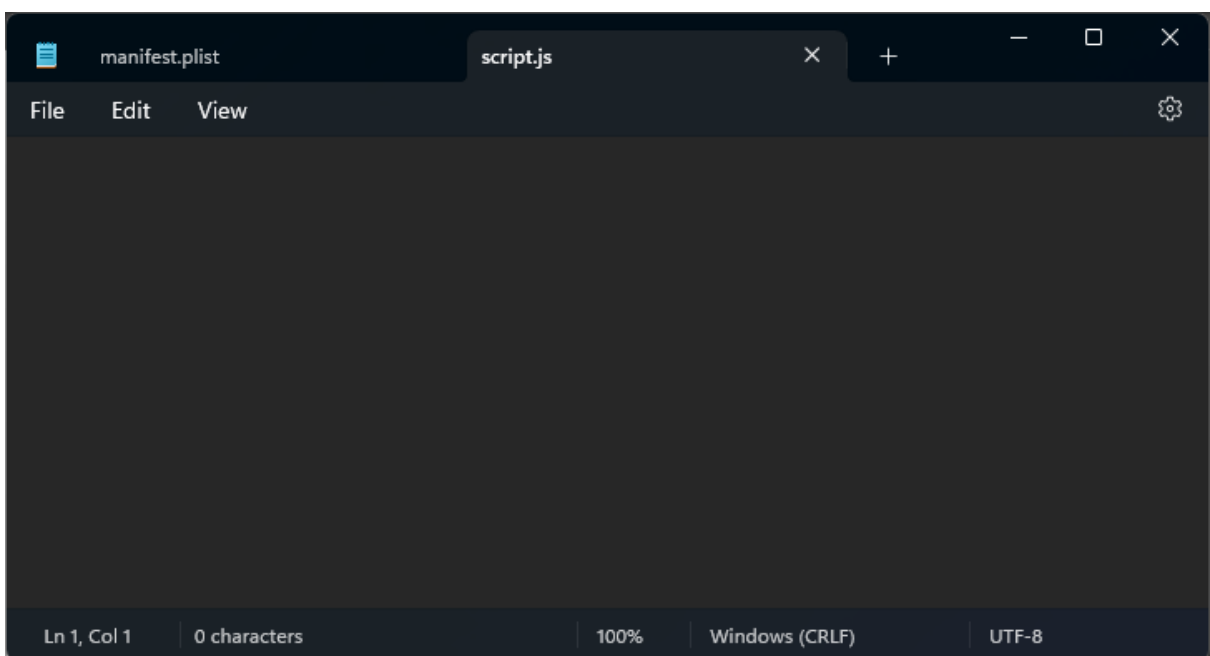
The important bits to change in this data is "myname" in the identifier and author fields. Make sure you keep the double quotes around the name, and make the semi-colon stays on the end of the line.

The other important bit is the "requires_oxps". This tells Oolite what expansions must be installed for this one to work. In our case, this whole OXP is dependant on PlanetFall, so we include its identifier here, as shown.

Next, we want to edit the script.js file. Again, I'm opening it in Notepad on Windows, and TextEdit on Mac.

Ln 1, Col 1   0 characters   100%   Windows (CRLF)   UTF-8
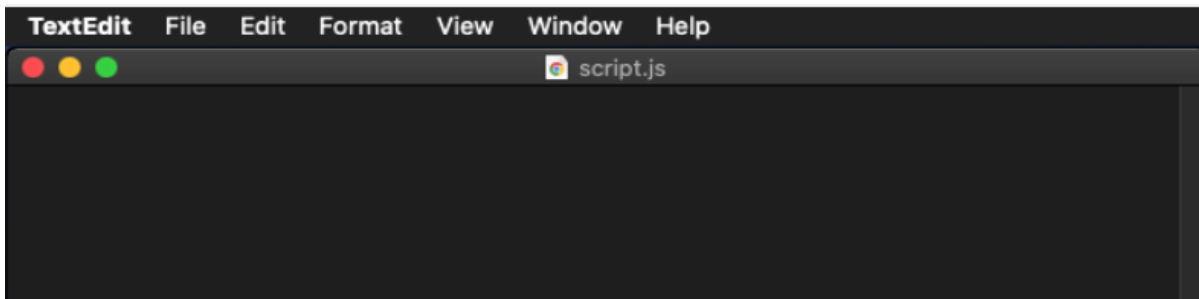
For the moment, we'll just put in the basics, and we'll add some actual code later. The minimum you should have is the following:

```
"use strict";
this.name = "SuperSystem";
this.author = "myname";
this.copyright = "2024 myname";
this.description = "Creates custom surface locations using PlanetFall2";
this.license = "CC-BY-SA-NC 4.0";
```

These lines should always go at the top of any script file in Oolite. The important bits are "use strict" and "this.name". The value in "this.name" should be unique. ie. No other script file in Oolite can have that name.

this.author, this.copyright, and this.description aren't strictly necessary, but they do help in identifying who wrote the script. this.license, however, is vital, as it tells future developers what you permit them to do with your code.
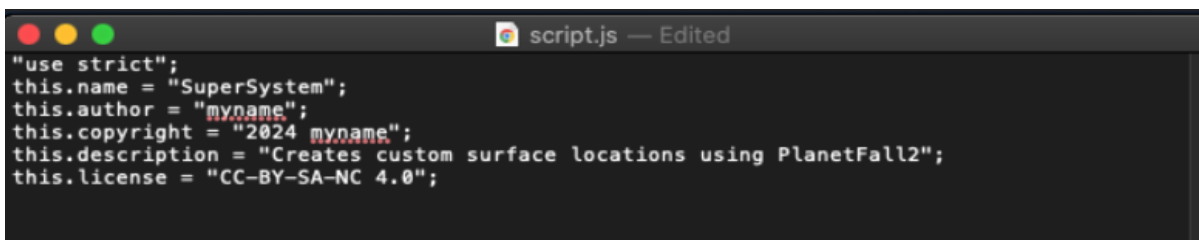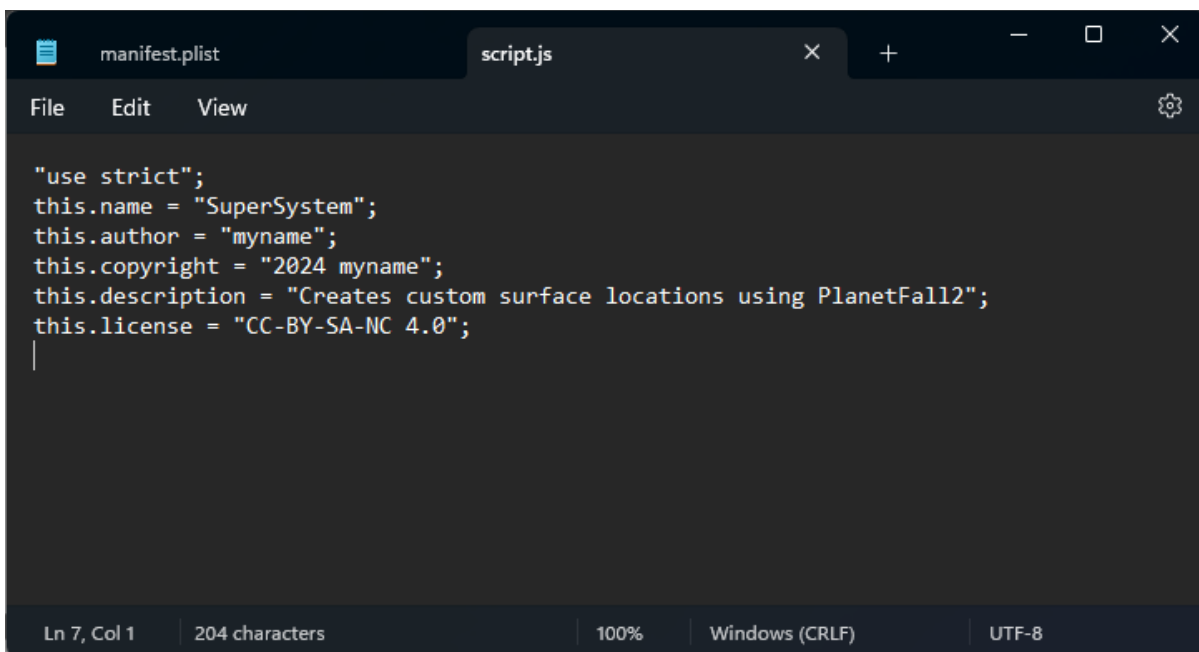
Note: From this point on, I'll only include the Windows screenshots, as the code is identical in both Windows and Mac versions.

This file, "script.js" is our world script. Oolite will pick this file up and include it with all the other world scripts from other files. Our world script is called "SuperSystem". There is another way to create a world script, but that's beyond the scope of this document. For the time being, it's enough that we have a world script, and Oolite will recognise it.

And that's it! If your files are saved, you could start Oolite now and it would recognise your OXP. However, given we haven't put any instructions in, it's a rather pointless mod. So, lets get into some meat and potatoes.

System definition

Using built-in locations

This might seem backwards, but we're going to start here, because if you don't need unique locations, this might be all you need. The way to tell if you will need a custom location is to check the built-in locations that come with PlanetFall2. For the main planet, there are 5 default options: **capitalCity**, **militaryBase**, **leisureComplex**, **factory**, and **dump**.

Using this.startUp world event

For our first system, let's say we want to set up 2 capitalCities, a leisureComplex and a factory. So, we need to start with a world **event** call "startUp", inside our world script. When certain things happen in the game, they trigger an **event**. Some events are specific to a particular object, like a ship or a station. Some, though, are global. These are world events, and the one we're going to use is called startUp. This event fires after the game has been loaded, but before the system is populated with planets, stations and ships. It's the perfect time to add in details we need.

```
this.startUp = function() {
    var pf = worldScripts.PlanetFall2;
    pf._locationOverrides["0 131"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure
Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
}
```

Let's step through this code one line at a time.

```
    this.startUp = function() {
```

This is how we hook into the world event. "startUp" is the name of the world event, and the "this." is a reference to the current script. This event needs to be a function with no parameters, which is why we have "= function()". The round brackets are where any parameters being passed into this function would be placed, but in this case, the startUp event doesn't have any, and so the brackets are empty. Finally, we have an open curly brace. This is the start of the code section for our function. You will notice there is a closing curly brace right at the end of the code. Coding in Javascript can be frustrating, especially when code is complex, because every opening bracket must have a closing one. So, every "(" has a ")", every "{" has a "}" and every "[" has a "]". And you can't close them in any order. If there is an "(", followed shortly thereafter by "[", you have to close the "]" before you can close the ")".

```
    var pf = worldScripts.PlanetFall2;
```

With this line, we are creating a variable called "pf", and we are telling the Javascript engine to create a referential link between it and the PlanetFall2 world script. If you were to look inside the PlanetFall2 OXP, and find the file "planetFall2_worldScript.js" you would find at the top of that file the following:

```
    this.name = "PlanetFall2";
```

That's the name of the world script, just as the name of ours is "SuperSystem".

By creating a reference to the PlanetFall2 world script, we can access some of the functions and properties of that script file. And that's what we're doing on the next line.

```
    pf._locationOverrides["0 131"] = {
```

With this code, we're referencing the dictionary object of the PlanetFall2 world script. A **dictionary** is a list of **keys** and associated **values**. Some examples of keys and values would be something like this:

My_Shopping_Basket["apples"] = 6

My_Shopping_Basket["oranges"] = 5

The "key" for my shopping basket is either "apples" or "oranges". Using the "apples" key, I can retrieve the associated value, which is 6. Using the "oranges" key, I can retrieve its associated value, which is 5. That's a dictionary.

The dictionary we'll be setting up is going to be a little more complex that that, however. But on the line of code above, we are telling the "_locationOverrrides" dictionary we are defining the key "0 131". Those two values, "0" and "131" are a reference to a galaxy (0, which is galaxy or sector 1 in the game – computers like to count from zero) and a single system, in this case Zadies.

As for the value we'll be putting into the _locationOverrides dictionary for key "0 131", it's another dictionary. We can tell that because a dictionary definition always starts with a curly brace {.

There are other ways to define dictionaries, but this will do for now. Once again, notice that our open curly brace has a closing brace "}", which is the second last line of our code sample.

The next line of code is:

```
main: [
```

Here, we are creating the first key in our new dictionary, a key called "main". Next to "main" is a colon ":". Because we are inside a dictionary definition at this point (ie, we have opened up the definition with the curly brace in the previous line "{") we switch from using "=" to define things, to ":". That's just how it is, I'm afraid!), the colon says, "the value for the key is the next thing you'll see".

And what you see is a square bracket "[". I know, brackets everywhere! In this case, the bracket is defining an **array**.

An array is just a list of things. It could be numbers (eg [30, 24, 80, 90, 23, 6]), or it could be text (eg ["hello", "world", "from", "space"]) or more complex things like ships or stations. The are certain properties of an array that make them useful. Firstly, you reference the elements of an array using an **index**. Each element has an index that starts at zero. So, using our number array example, the item at index 0 is 30, the item at index 3 is 90.

Fortunately, we don't need to access the elements of the array in our code. We only need to define the content.

The closing bracket for this array is the third last line of our code sample.

I'm going to cover a couple of lines here at once, because it makes understanding it easier.

```
    {
        roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
        names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure
Centre)", "Bob's Place (Automobile Factory)"],
    },
```

This code defines the content of our array, and in this case, it's another **dictionary** (see that open curly brace at the start, and the closing one at the end?) Dictionaries have **keys** and **values**. In our code, we have two keys: "roles" and "names". The value for both of these is an **array** (see the open square bracket "[", and the closing one "]" for each value?).

With the "roles" key, this is where we tell PlanetFall2 what types of locations we want to create. The order of items isn't important in itself, with the only stipulation being that the equivalent name (defined in the array value for the "names" key), lines up. That is, if the first element of our roles array is a capitalCity, the first element of the names array should be the name for that capitalCity.

Note also at the end of each array, there is a comma following the closing square bracket. Because we are still inside the dictionary definition, individual key/value pairs are separated by a comma. Strictly speaking, the

comma at the end of the "names" array isn't necessary, because it's the last key/value pair in the dictionary, but it helps to prevent future issues if we ever decide an extra key/value pair is needed.
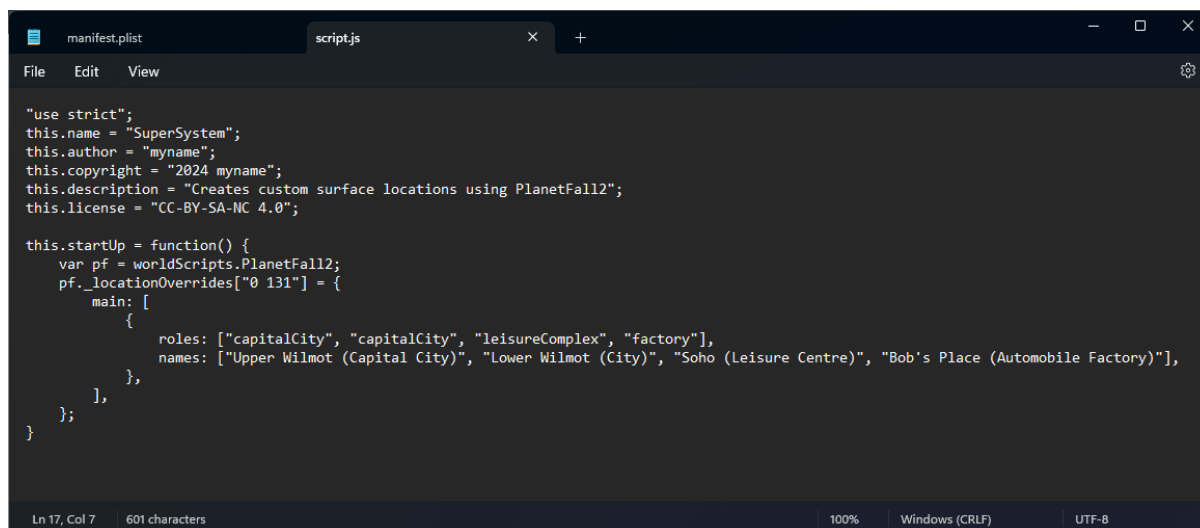
We have now created the two key/value pairs in our dictionary, so we can close the dictionary with a "}".

You might remember that the dictionary we just created was inside an array. Elements in an array are separated with a comma as well, and so there is a final "," after we close the dictionary.

From this point on, we are simply closing all the things we opened. The array for "main" is closed with a "]", then the dictionary we are creating for system "0 131" is closed. Because this is the end of the dictionary definition, and technically this is a code line (rather than a dictionary entry), we add a semi-colon ";" to the end of the line.

Concerning semi-colons and Javascript: As an aside, semi-colon terminators for code lines are not actually required. You can leave them out, and the code will still work. However, I recommend getting used to adding the ";" to the end of every code line because makes reading the source code easier, particularly if you regularly switch between different languages, many of which use the ";" as a line terminator. If you put a ";" at the end of the code line, that is declaring to any future reader that this is the end of that code line. It's like having a full stop at the end of a sentence. Yes, sentences can be read successfully without them, but it helps your readers keep track of where things start and finish. Also, while they are unnecessary in JS files, they are crucial in OpenStep plist files. So, it's probably easier to go with the default of always including them, than working out when you don't need to.

And the last "}" is the end of our function "startUp".

And that's it. That's the most basic version of creating a custom system definition.

Custom locations

Using shipdata.plist

But it's fairly boring, and not much more that what you would get by default with PlanetFall2. So, let's create a new definition for another planet, and create a unique location for it, that will only be visible in that system. To do that, we first need to create a new file, called "shipdata.plist". This is another text file, and it's put in the same folder as "script.js", in the "Config" folder.
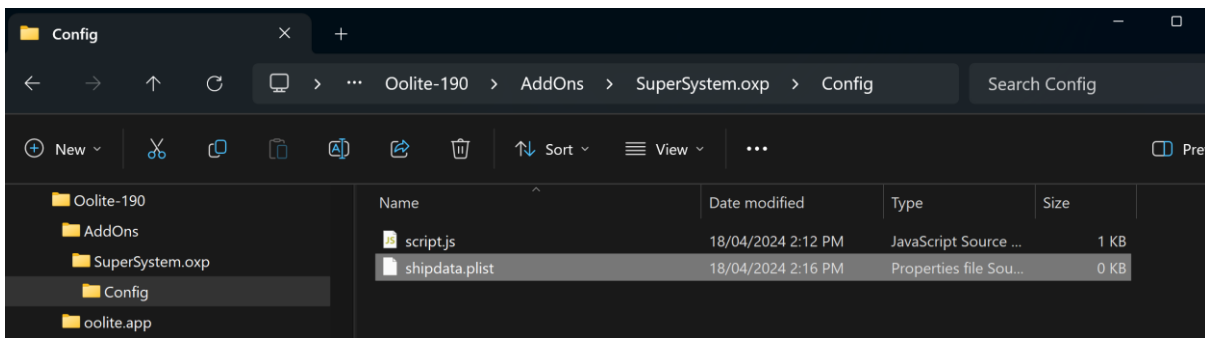
shipdata.plist is where ship entities are defined for Oolite. If you see it flying around in the game, it will be defined in a shipdata.plist file somewhere. (There are exceptions, but let's just roll with that definition for now).

The shipdata.plist is a dictionary file. That is, the content of the file is one big dictionary. Each shipdata.plist must have an opening and closing curly brace:
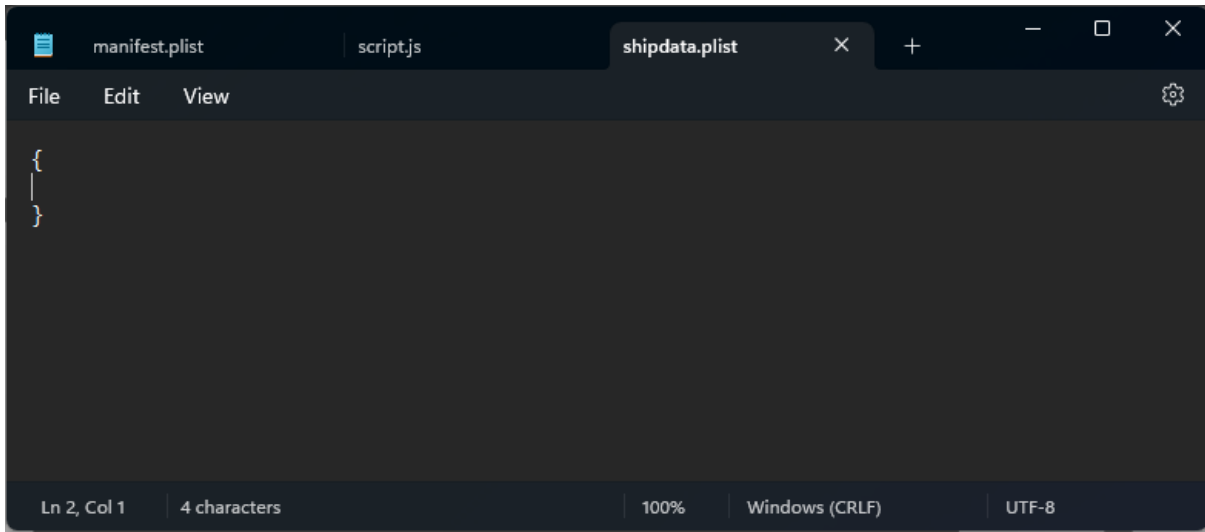
A dictionary is a list of key/value pairs. So, the first thing we need to create is a ship key.

Every ship key must be unique. If Oolite detects a duplicate, one will be overwritten with the other, but there is no way to know which one "wins" and which one "loses". By keeping our ship keys unique, we prevent that issue from taking place.
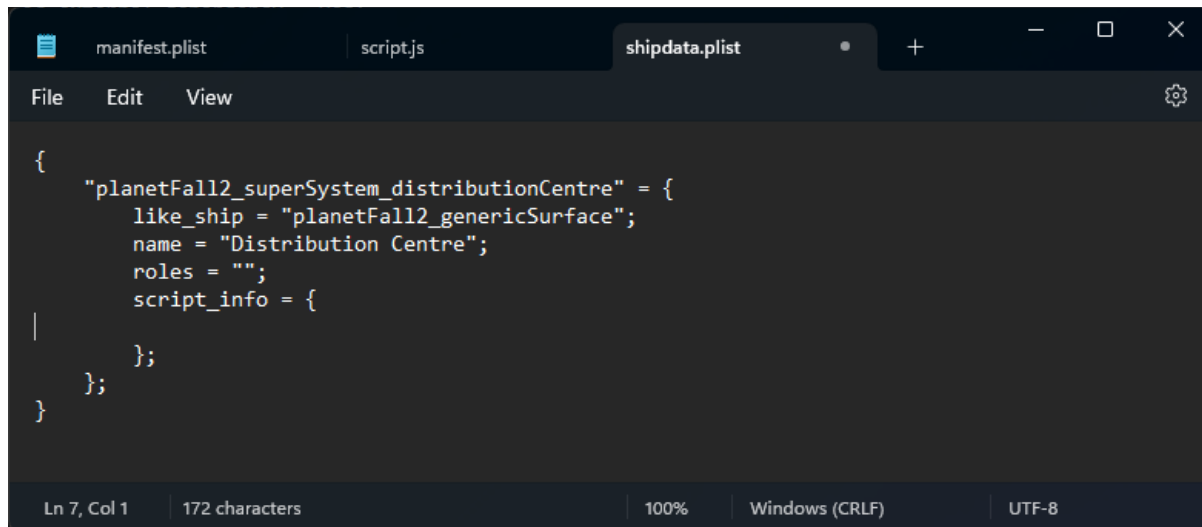
For the new system, we're going to create a "Distribution Centre". We'll make the key "planetFall2_superSystem_distributionCentre".
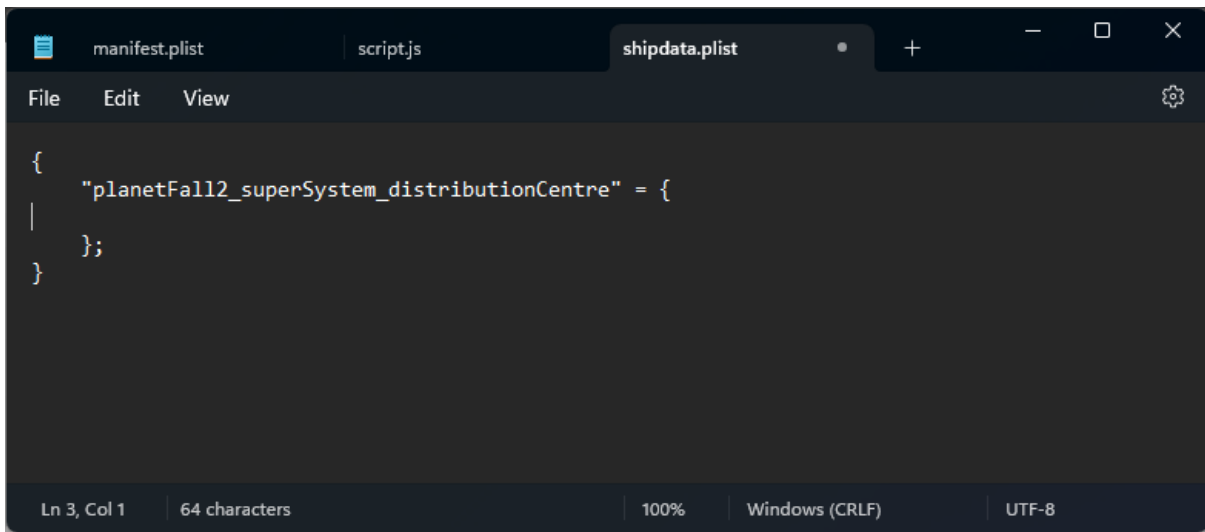
In the screenshot above, I've created the key, followed by an "=" (because we're in a plist file, and not in Javascript code, assigning values to dictionary keys is done with an "=", not a ":"). Then, because each ship definition is itself a dictionary, we open a new curly brace, and to make sure we do things properly, we add a

closing brace and ";" signals the end of the individual ship definition (again, because we're in a plist file, and not Javascript code, individual items are separated with a semi-colon, not a comma).

That's our basic structure in place. There are a few things that are required for our definition to integrate nicely with PlanetFall2.
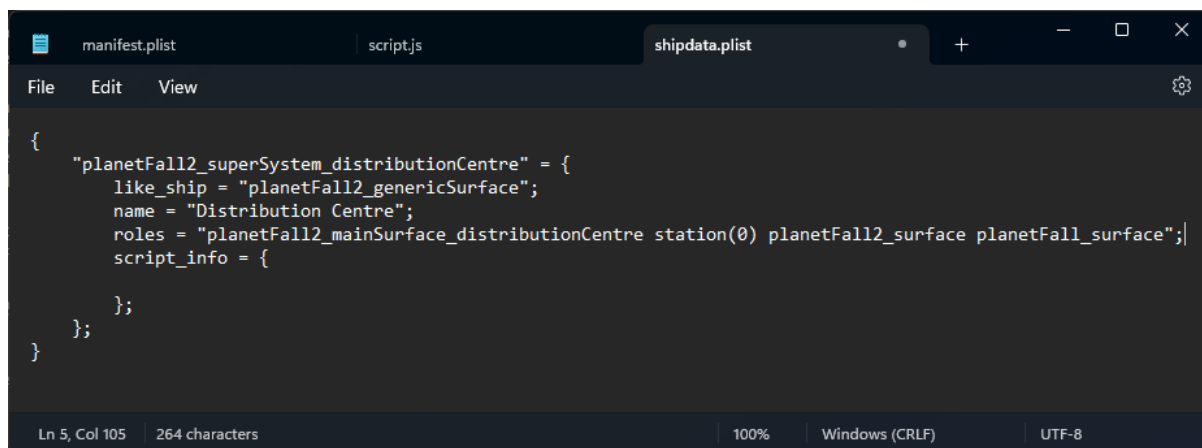

The first requirement is the "like_ship" entry. This links our definition to the generic surface definition in PlanetFall2 and enables us to pick up all its default values, meaning we don't have to repeat them all here. All we need to do is define the things that will be different.

The first difference is, naturally enough, the name, which is descriptive enough. This name, though, isn't the one that we'll see in game. This is just the generic name.

Next, we have "roles", and "script_info", both of which are empty in the screenshot above. The values we put into these keys will control a lot of what we can do with our new destination.

Because we need a unique role for each shipdata entry, we added "planetFall2_mainSurface_distributionCentre" as the first role. That role is what we use later when we're adjusting our system configuration. Notice it is not the same as the ship key: the centre bit is "mainSurface", while the key has "superSystem".

All the following roles are required. "station(0)" is simply to tell Oolite that what we are defining is a station, something the player can dock at. "planetFall2_surface" and "planetFall_surface" are so that other OXP's can find PlanetFall destinations easily.

There are other keys we could use, depending on what we were trying to achieve. But because we only want to use this ship key in one place, those are all the roles we need.

Finally, there is the script_info dictionary. This is where we can pass custom information from our definition here, and onto an actual object when it is created in Oolite. Each ship created that has a "script_info" dictionary

defined in its shipdata, will have a property called "scriptInfo" in Oolite, and the properties of that will match the key/value pairs we define here.

So, what do we want to define?

```
{
    "planetFall2_superSystem_distributionCentre" = {
        like_ship = "planetFall2_genericSurface";
        name = "Distribution Centre";
        roles = "planetFall2_mainSurface_distributionCentre station(0) planetFall2_surface planetFall_surface";
        script_info = {
            telescope = 0;
            beacon_code = "D";
            allow_f3 = false;
            market_type = 3;
            market_key = "planetFall2_mainSurface_capitalCity";
            landing_image_set = "factory";
        };
    };
}
```

Ln 12, Col 41   487 characters   100%   Windows (CRLF)   UTF-8

There are a few things we need to do straight away.

telescope = 0;   This tells the "Telescope" OXP to ignore this entity when scanning the system. The objects that get created by PlanetFall should be invisible. That is, they need to be present in the game, because you have to dock with them, but they should not be visible anywhere. The location where they are created should be far enough away from anywhere to prevent accidental discovery, but it's better to tell Telescope to "mind it's own business" for our entity.

beacon_code = "D ";   Theoretically, we could put the beacon code directly into the ship data. But what that would means is that our "invisible" dock would turn up on the space compass, which we don't want. Instead, we want to pass on the code to the object that will be created near the surface of the planet. And this is where we do that.

allow_f3 = false; This instruction tells PlanetFall to disable access to the F3 Equip Ship screen. If you wanted to permit access, you would simply make this "true".

market_type = 3;   This tells PlanetFall the type of core market that we want to use for our station. There are 3 default markets available: Type 1 is if we want to have a copy of the main station market. Or if we want to utilise the "market_definition" property of shipdata (that is, we want to define our own market using the built-in facilities of Oolite). Type 2 is using the method employed in the "Stations For Extra Planets" mod. This adjusts prices and quantities based on how far away the station is from the main station. And Type 3 is how the original PlanetFall (ie version 1.51 and prior) defined the markets, before the markets were broken with Oolite 1.84. We've set type 3 in our example, and because of that, we need an extra key/value pair.

market_key = "planetFall2_mainSurface_capitalCity"; This tells PlanetFall what key to use to lookup the market with. In this instance, we've told it to use the market definition from the capital city.

landing_image_set = "factory"; This tells PlanetFall which landing image set to use whenever the player lands at that location. In this instance, we've told it to use "factory", which is the set used for factories on main planets and extra planets. To re-use one of the default image sets, you can use one of these keys:
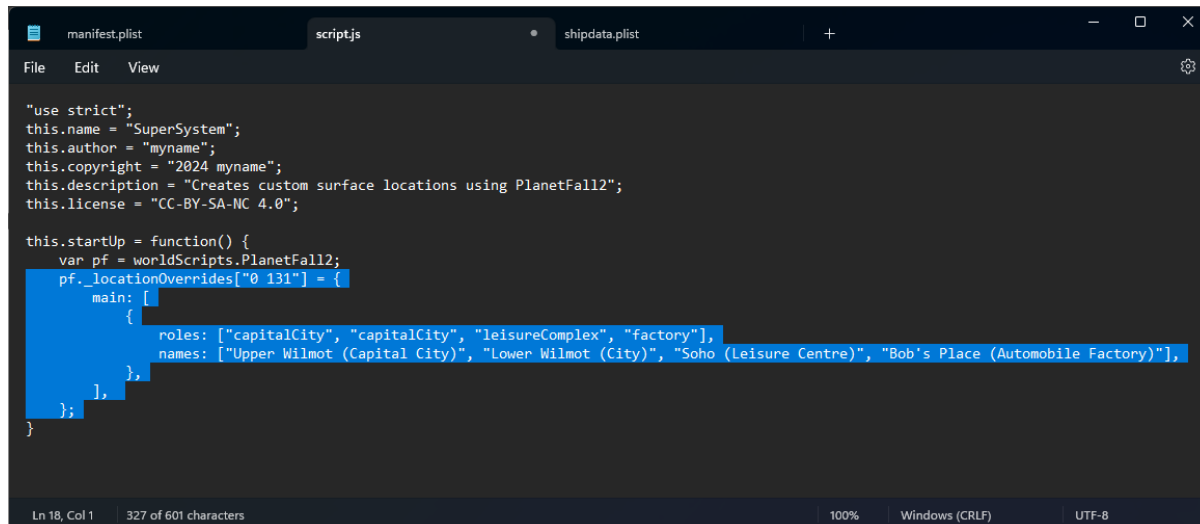
> **city**: for main planet capital cities.
> **military**: for main planet and extra planet military bases.
> **leisure**: for main planet and extra planet leisure centres.
> **factory**: for main planet and extra planet factories.
> **dump**: for main planet and extra planet rubbish dumps.
> **colony**: for extra planet colonies.
> **moon_dome**: for moon colony domes.

moon_leisure: for moon leisure domes.
moon_research: for moon research complexes.
moon_mine: for moon mines.
moon_factory: for moon robot factories.
moon_wasteland: for moon wastelands.

That's it for the shipdata.plist file. Now we can have another look at our script.js code.

Code for script.js

What we want to do is copy the exist definition we made, and paste it immediately after the existing one, so we don't have to type as much.
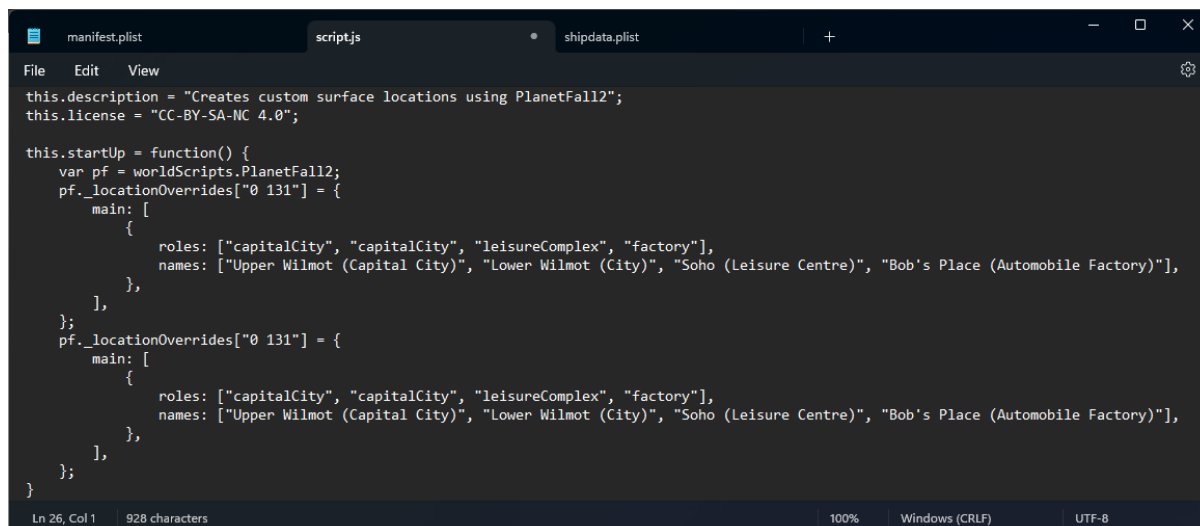

Highlight the code as indicated in the screenshot above. Copy it, and then paste it immediately after that code (ie after the "};" and before the final "}"). It should look something like this:

At the moment, this code would do nothing. We're essentially just setting up Zadies twice. So, let's pick another planet, Esusti, planet ID 62, which is one jump away from Zadies.
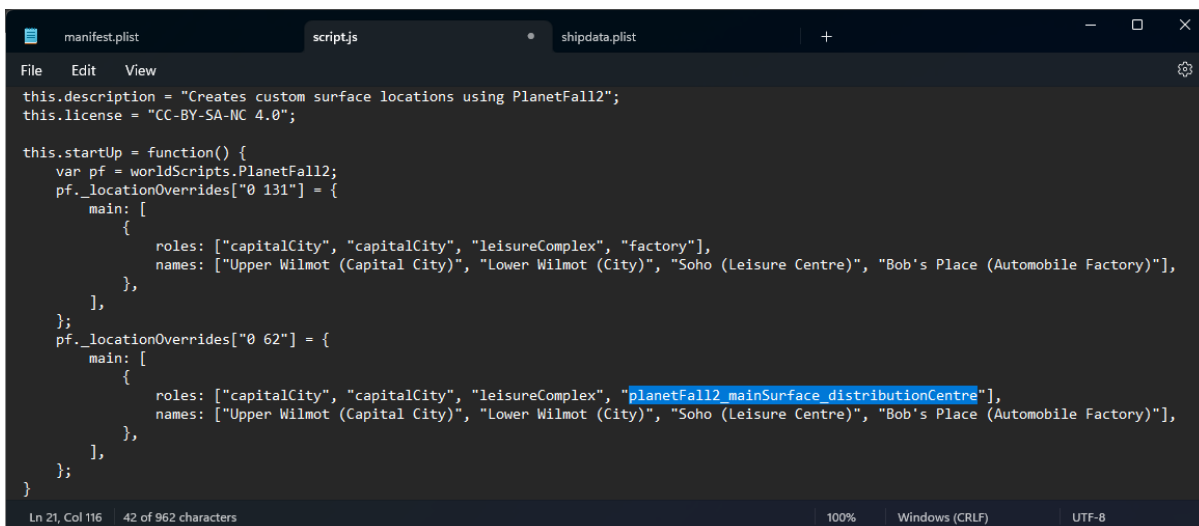
```
manifest.plist          script.js          ●  shipdata.plist        +

File   Edit   View
this.description = "Creates custom surface locations using PlanetFall2";
this.license = "CC-BY-SA-NC 4.0";

this.startUp = function() {
    var pf = worldScripts.PlanetFall2;
    pf._locationOverrides["0 131"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
}
```

Ln 18, Col 32   2 of 927 characters   100%   Windows (CRLF)   UTF-8

Now we have two definitions. But right now, they're identical. Anyone travelling between Zadies and Esusti would get a bit of deja vu. "Didn't I just land at Upper Wilmot?"

So, let's fix this up. First, we'll change the factory to the **role** of our new distribution centre.
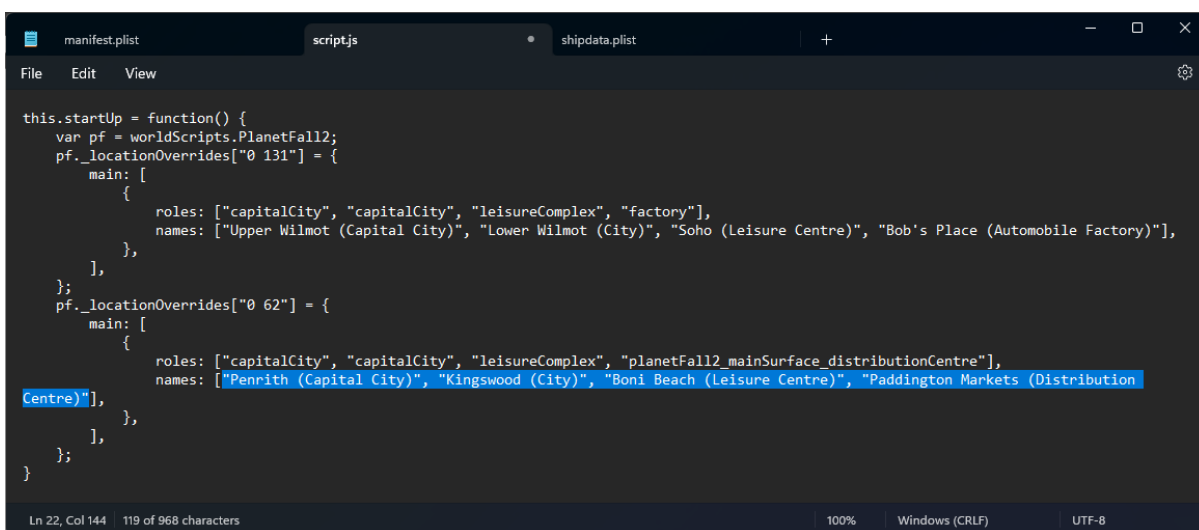


```
manifest.plist          script.js          ●  shipdata.plist        +

File   Edit   View
this.description = "Creates custom surface locations using PlanetFall2";
this.license = "CC-BY-SA-NC 4.0";

this.startUp = function() {
    var pf = worldScripts.PlanetFall2;
    pf._locationOverrides["0 131"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "planetFall2_mainSurface_distributionCentre"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
}
```

Ln 21, Col 116   42 of 962 characters   100%   Windows (CRLF)   UTF-8

And then we can give all the places unique names.


```
manifest.plist          script.js          ●  shipdata.plist        +

File   Edit   View

this.startUp = function() {
    var pf = worldScripts.PlanetFall2;
    pf._locationOverrides["0 131"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
                names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "planetFall2_mainSurface_distributionCentre"],
                names: ["Penrith (Capital City)", "Kingswood (City)", "Boni Beach (Leisure Centre)", "Paddington Markets (Distribution Centre)"],
            },
        ],
    };
}
```

Ln 22, Col 144   119 of 968 characters   100%   Windows (CRLF)   UTF-8

And that is how quickly we can add new custom definitions. But what if we want some different landing images for our newly defined distribution centre? Let's get into that.
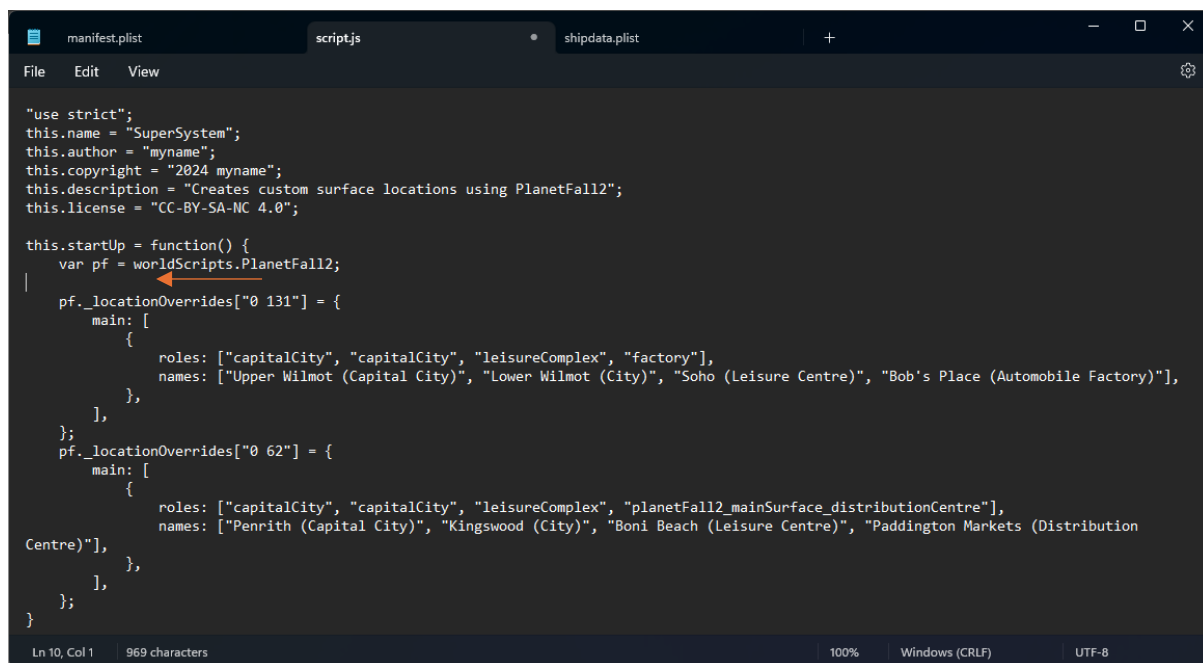
Landing Image Sets

I'm going to have to assume you have some images you can use for this. For the sake of argument, lets call them "planetFall2_dist_1.png", "planetFall2_dist2.png", and "planetFall2_dist3.png".

First you need to create an "Images" folder in your OXP folder.
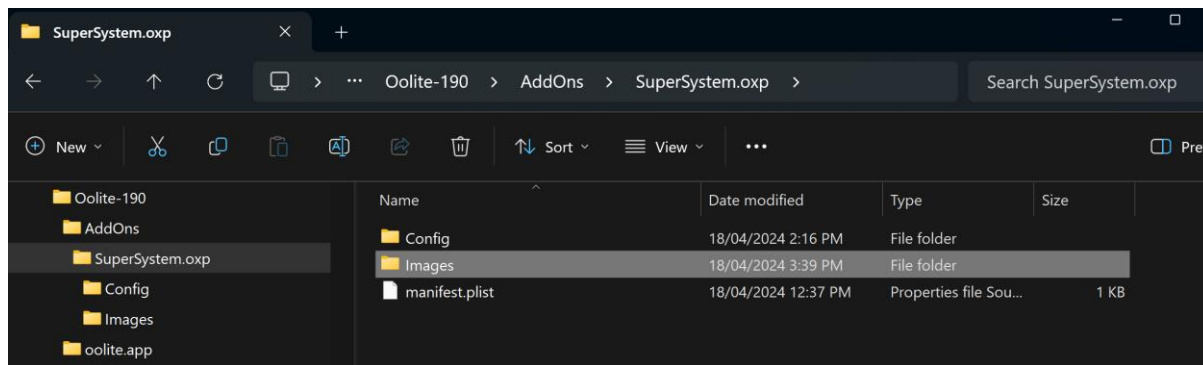
Put your images into that folder.

Next, we need to tell PlanetFall what to do with those images. Open the script.js file again and add a new line immediately after the "var pf = worldScripts.PlanetFall2;" line:

What we need to do is to add a new key/value pair to the "_landingImages" dictionary in PlanetFall2. We start by typing this:

```
pf._landingImages["distCentre"] = [];
```

That sets up the array we'll need for entering our image filenames. Then it's just a matter of typing the names in:

```
pf._landingImages["distCentre"] = ["planetFall2_dist1.png", "planetFall2_dist2.png",
"planetFall2_dist3.png"];
```


That's all we need to do in our script file. Save that, and then open the shipdata.plist file. In this file, we need to change the landing_image_set from "factory" to the new key we defined, "distCentre":

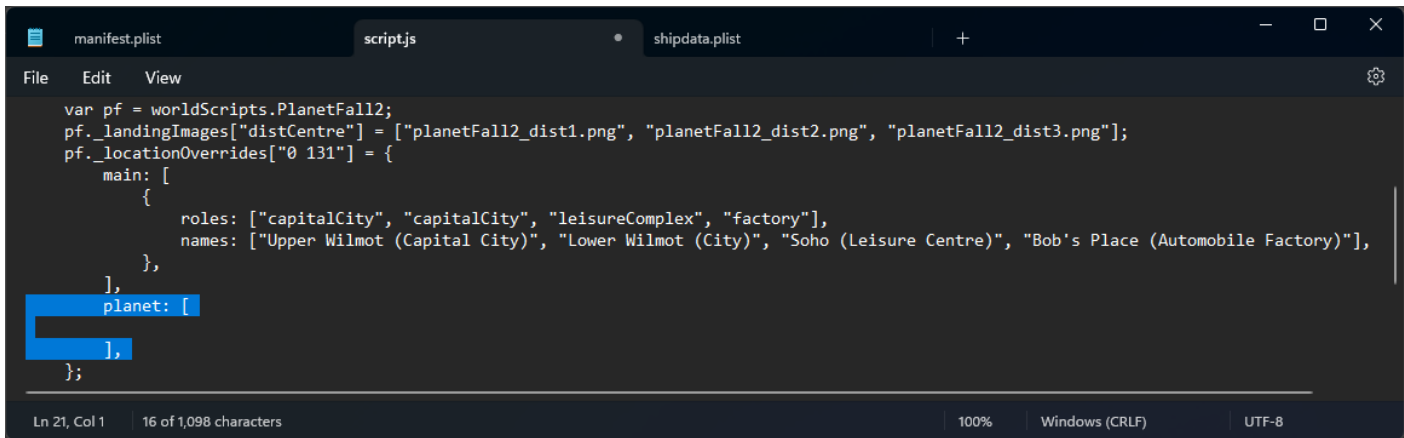And that's it. We have a new custom planet configuration, with a unique location and custom landing images.

Note: With landing images, PlanetFall2 will attempt to select the same image for the same location every time. So, while we have 3 different images in the image set, we are only using the distribution centre definition once, and so only 1 of the images will ever be selected, as PlanetFall will choose the same one each time.

Extra Planets and Moons

We have now successfully defined the main planet for two systems. But what about extra planets and moons? Time to head back to our script.js file. Lets look at our code for Zadies again:

```
pf._locationOverrides["0 131"] = {
    main: [
        {
            roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
            names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's
Place (Automobile Factory)"],
        },
    ],
};
```

To expand this definition to cover extra planets, we need to add a new key, called "planet". It will follow immediately after the "main" key.

The "planet" key has an array value (remember, square brackets). Inside this array we're going to create the same structure as we used for the main planet:

```
    main: [
        {
            roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
            names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
        },
    ],
    planet: [
        {
            roles: [],
            names: [],
        },
    ],
};
```
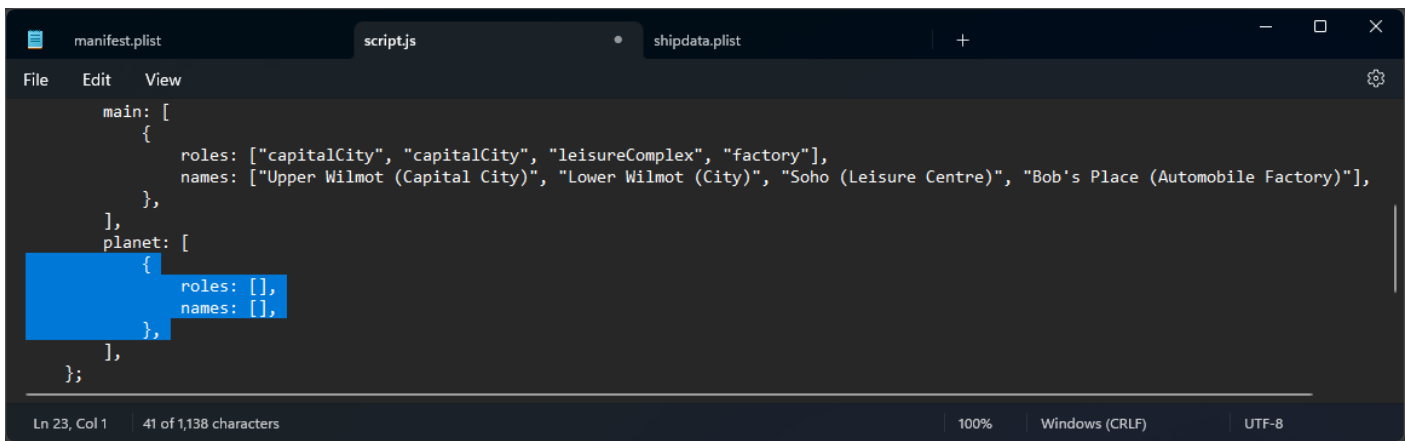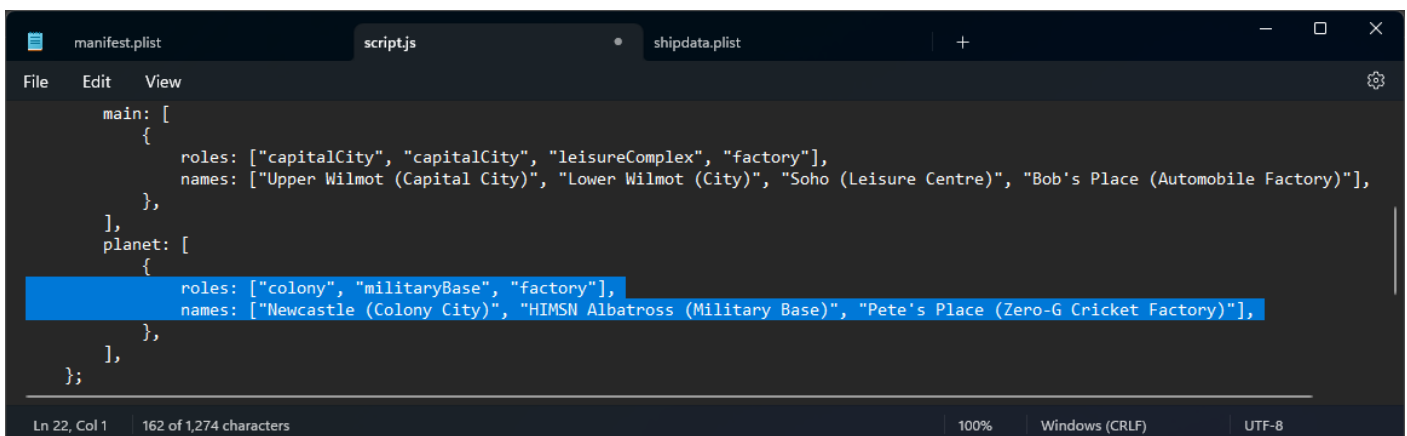
Here, we've created the first element of our array, which is a dictionary with two keys: roles and names. Both of these arrays are empty in the screenshot.

Now we can add in the roles and names for the first extra planet that gets defined in Zadies. Depending on how you have setup SFEP, the number of extra planets could be variable. However, for this example, we'll assume there are two extra planets.

The built-in default roles we can use for extra planets are: **colony**, **militaryBase**, **leisureComplex**, **factory**, and **dump.** The list for extra planets is almost identical to main planets, the only difference being the first one.
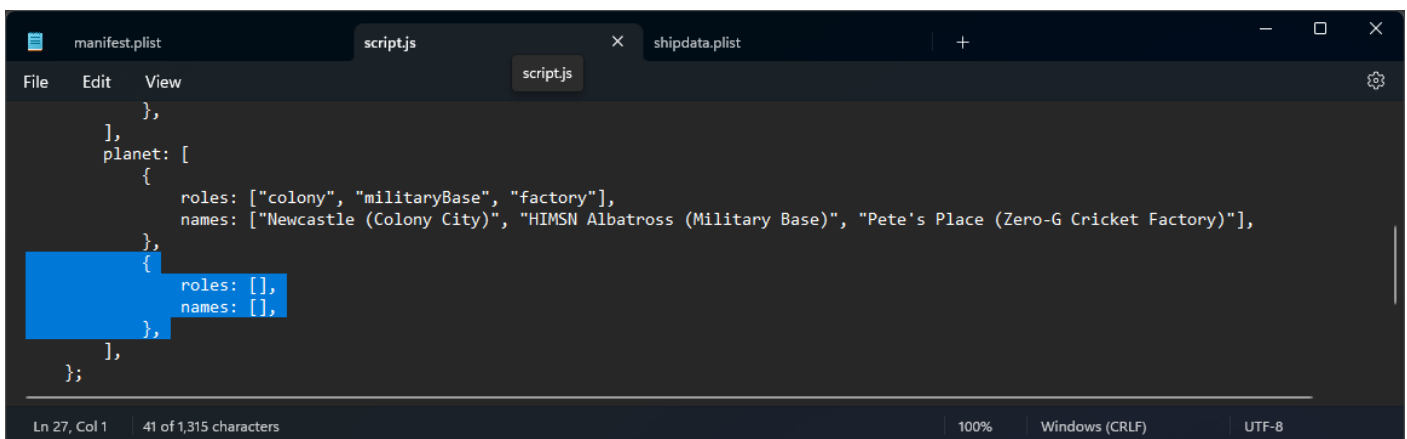
So, let's set up the first extra planet with some default roles.

```
    main: [
        {
            roles: ["capitalCity", "capitalCity", "leisureComplex", "factory"],
            names: ["Upper Wilmot (Capital City)", "Lower Wilmot (City)", "Soho (Leisure Centre)", "Bob's Place (Automobile Factory)"],
        },
    ],
    planet: [
        {
            roles: ["colony", "militaryBase", "factory"],
            names: ["Newcastle (Colony City)", "HIMSN Albatross (Military Base)", "Pete's Place (Zero-G Cricket Factory)"],
        },
    ],
};
```

In this example, we've set up three locations, and created three names to go with them.
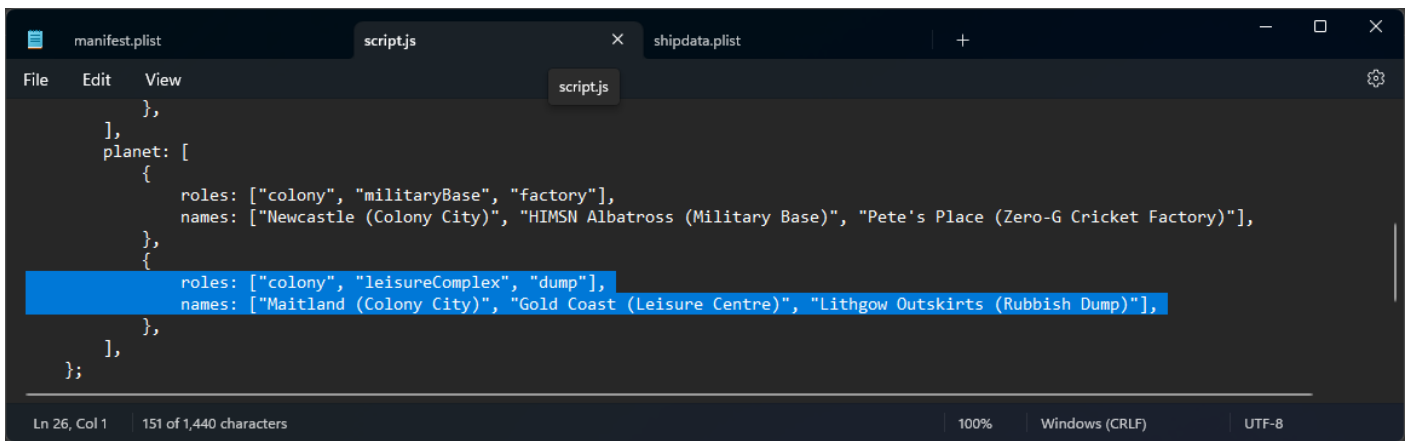
Now let's set up the second extra planet.

```
        },
    ],
    planet: [
        {
            roles: ["colony", "militaryBase", "factory"],
            names: ["Newcastle (Colony City)", "HIMSN Albatross (Military Base)", "Pete's Place (Zero-G Cricket Factory)"],
        },
        {
            roles: [],
            names: [],
        },
    ],
};
```

The comma at the end of the first extra planet definition is the separator between elements in our array, so we can add the new definition below it easily. Notice I've left the comma on the end of the new definition, so that if we decide in future to add another extra planet definition, we won't have to remember to add the separator.

We can now continue with our second extra planet definition.

```
        },
    ],
    planet: [
        {
            roles: ["colony", "militaryBase", "factory"],
            names: ["Newcastle (Colony City)", "HIMSN Albatross (Military Base)", "Pete's Place (Zero-G Cricket Factory)"],
        },
        {
            roles: ["colony", "leisureComplex", "dump"],
            names: ["Maitland (Colony City)", "Gold Coast (Leisure Centre)", "Lithgow Outskirts (Rubbish Dump)"],
        },
    ],
};
```
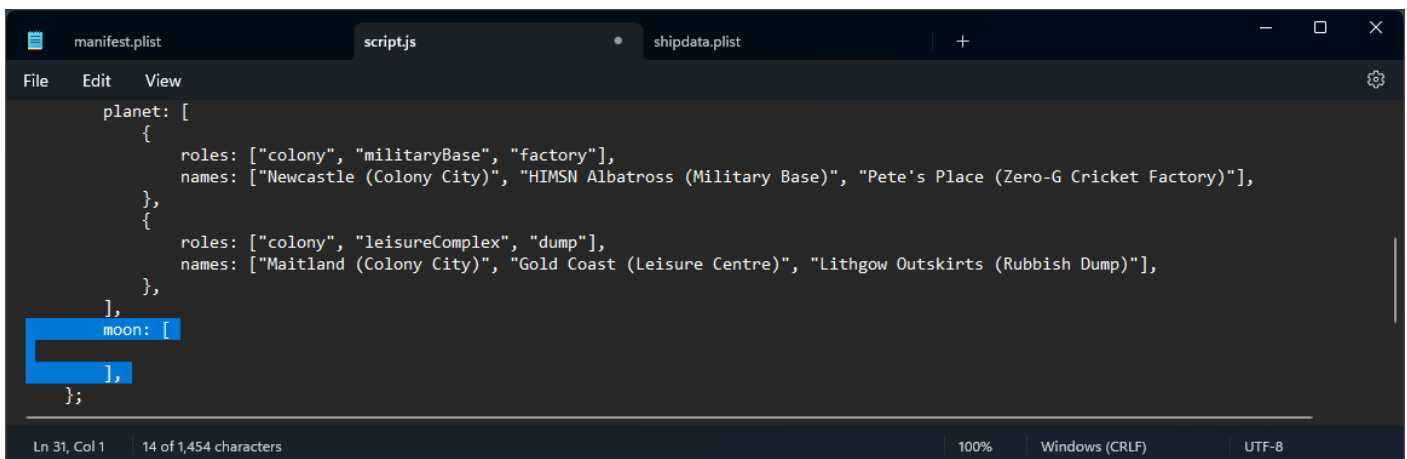
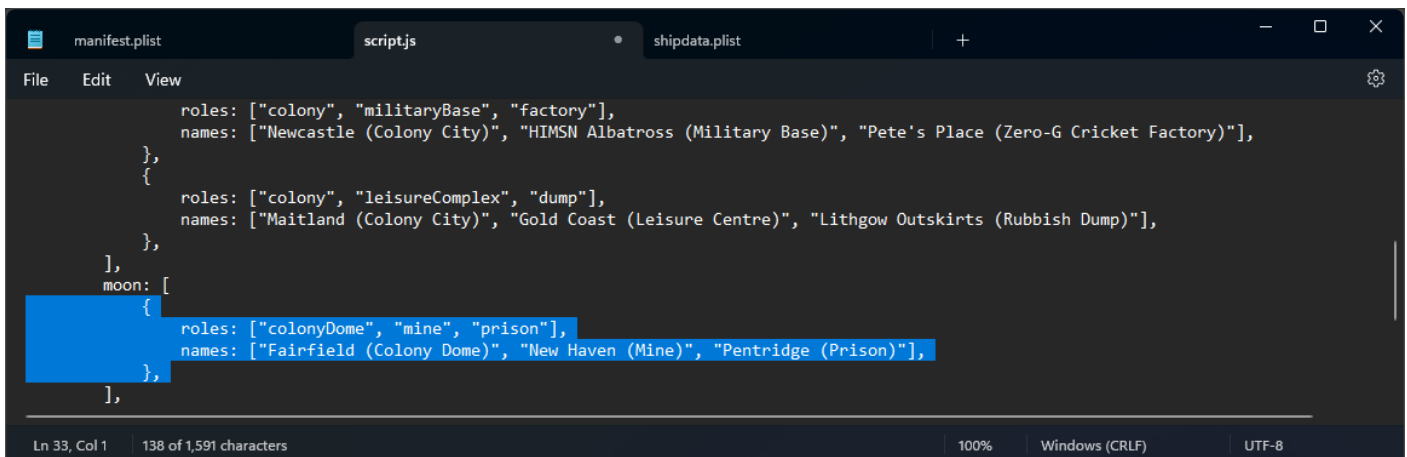Ln 26, Col 1   151 of 1,440 characters   100%   Windows (CRLF)   UTF-8

We now have two extra planet definitions. If your Zadies system populated with only one extra planet, the second definition would not be used. If, however, Zadies populated with three extra planets, the first two would use these custom definitions, but the third one would get default random landing sites.

Adding custom moon definitions is simply a matter of creating a new key: "moon", below the "planet" entry.


```
    planet: [
        {
            roles: ["colony", "militaryBase", "factory"],
            names: ["Newcastle (Colony City)", "HIMSN Albatross (Military Base)", "Pete's Place (Zero-G Cricket Factory)"],
        },
        {
            roles: ["colony", "leisureComplex", "dump"],
            names: ["Maitland (Colony City)", "Gold Coast (Leisure Centre)", "Lithgow Outskirts (Rubbish Dump)"],
        },
    ],
    moon: [

    ],
};
```

Ln 31, Col 1   14 of 1,454 characters   100%   Windows (CRLF)   UTF-8

We then repeat the process for adding the dictionary elements of the array and putting the roles and names into each one. The built-in default roles we can use for moons are: **colonyDome**, **leisureDome**, **mine**, **prison**, **researchComplex**, **factory**, **wasteland**.


```
            roles: ["colony", "militaryBase", "factory"],
            names: ["Newcastle (Colony City)", "HIMSN Albatross (Military Base)", "Pete's Place (Zero-G Cricket Factory)"],
        },
        {
            roles: ["colony", "leisureComplex", "dump"],
            names: ["Maitland (Colony City)", "Gold Coast (Leisure Centre)", "Lithgow Outskirts (Rubbish Dump)"],
        },
    ],
    moon: [
        {
            roles: ["colonyDome", "mine", "prison"],
            names: ["Fairfield (Colony Dome)", "New Haven (Mine)", "Pentridge (Prison)"],
        },
    ],
```

Ln 33, Col 1   138 of 1,591 characters   100%   Windows (CRLF)   UTF-8

Once again, this moon definition would be allocated to the first moon created in Zadies. If we don't add any more definitions, any other moons will receive a random selection.
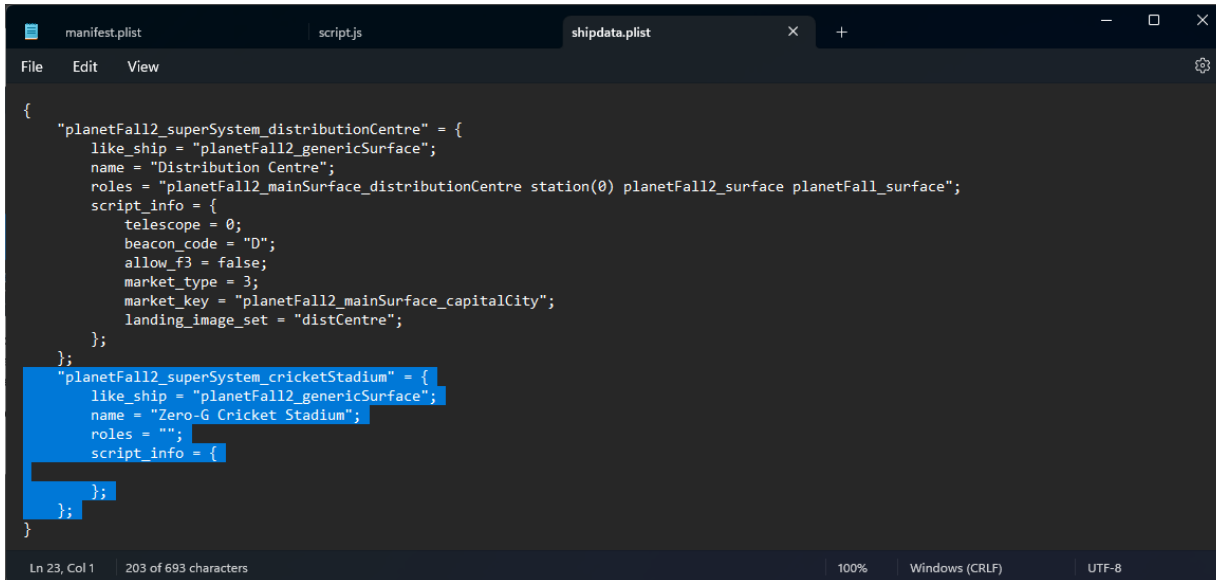
Alternatives

Adding new locations to the default pool

What we have covered so far is how to create a completely custom system. But what if, instead, you wanted to add a new item into the pool from which PlanetFall will pick when creating a randomly defined system? Can the system handle that?
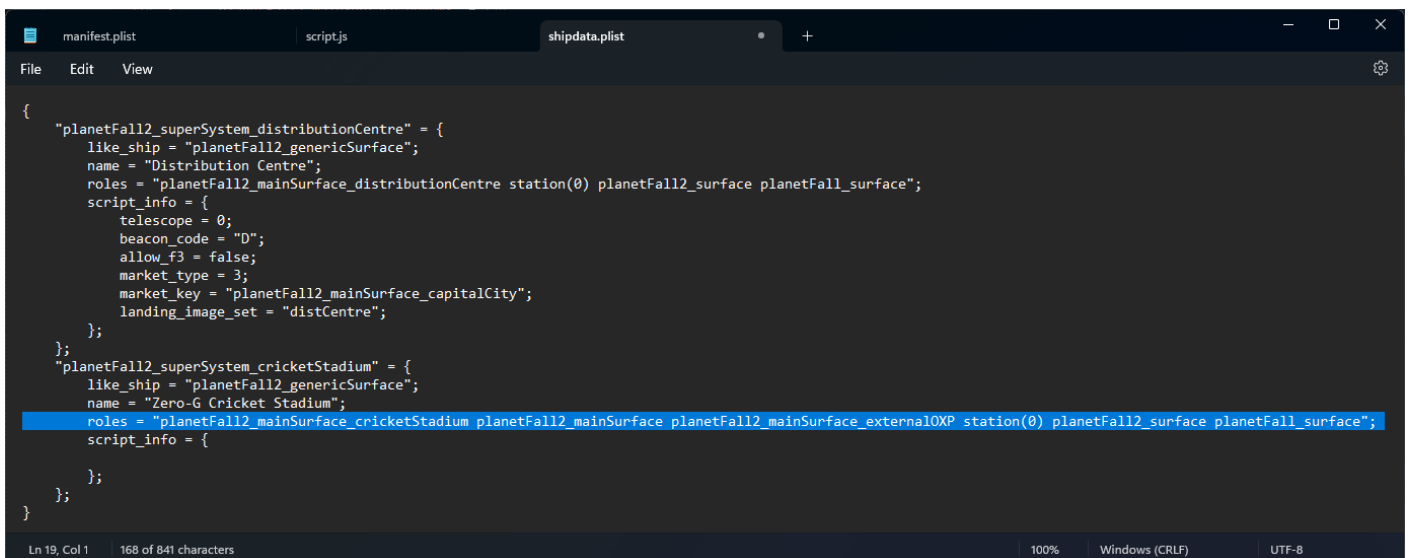
It sure can.

You would start with your shipdata.plist file. We will need to create a new entry for this:

Here, we've set up a new entity for a cricket stadium. We've picked up the basic entries as before. Now we need to assign roles. And it's here that things change a little.

The roles we've assigned are:

planetFall2_mainSurface_cricketStadium: the unique role for this entity.

planetFall2_mainSurface: this role will mean this entity will be picked up when PlanetFall2 selects entities to include on the main planet.

planetFall2_mainSurface_externalOXP: this role will mean if the system override flag is set, this entity will be included when PlanetFall2 selects entities for the main planet (if the override role has not been changed).

station, planetFall2_surface, planetFall_surface: required roles.

The important role to include is "planetFall2_mainSurface", as that is the default role PlanetFall uses when selecting possible landing sites.

But we need to add some script_info entries as well to round out the definition.

The first 6 entries are the same ones we used before. The only difference is that "market_type" has been set to "0", which means "No market at all".

The next four entries control how frequently this landing site will be spawned and named.
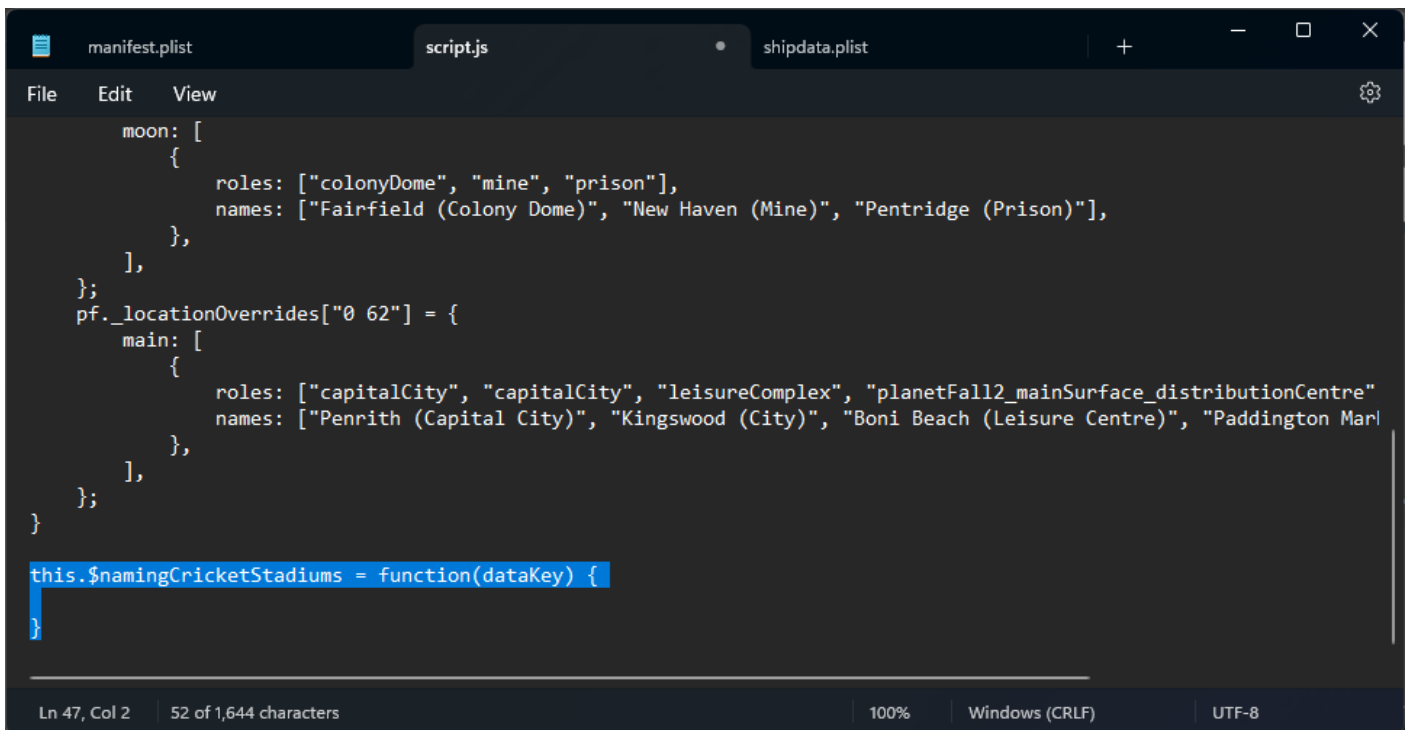
spawn_chance = 0.1; This means there will be a 10% chance the item will be spawned. The value can be lower (eg 0.01 would be a 1% chance, all the way up to 1.0, which would be a 100% chance).

seed = 2345; This is a number value that tweaks how the random number generator is "tuned", shall we say. Our code will be using a specific function that will create the same random number in any given system, but will be different in every system. This is perfect for our requirements, because we want the surface locations to be the same every time we visit the system. But, without a seed value, that means the function would return the same value every time we used it in the same system. What the seed value does is give you another random number that is different to the "no-seed" random number, but again, will always be the same in the current system, and different in every other system. This allows us to uniquely randomise this particular entity.

naming_function = "$namingCricketStadiums"; This defines the world script function name that will be used to give the entity a unique name. Without defining this, the entity would end up being named "Zero-G Cricket Stadium", which isn't bad, but if you want to add a bit more flavour, we can customise it with a call to this function.

naming_ws = "SuperSystem"; This defines the world script where the function named above can be located.

Let's set up that function now.

File   Edit   View

```
        moon: [
            {
                roles: ["colonyDome", "mine", "prison"],
                names: ["Fairfield (Colony Dome)", "New Haven (Mine)", "Pentridge (Prison)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "planetFall2_mainSurface_distributionCentre"
                names: ["Penrith (Capital City)", "Kingswood (City)", "Boni Beach (Leisure Centre)", "Paddington Marl
            },
        ],
    };
}

this.$namingCricketStadiums = function(dataKey) {

}
```

Ln 47, Col 2   52 of 1,644 characters   100%   Windows (CRLF)   UTF-8

The naming function is an example of a function that has a parameter being passed into it, in this case called "dataKey". "dataKey" will contain the ship dataKey (ie. The key used in shipdata.plist to uniquely identify each ship).

What we do with "dataKey", and indeed, this entire function, can be as simple or as complex as you can make it. Let's keep it simple. We'll make up a list of 5 names, and the function can pick one at random, although we want it to pick the same one each time for a particular system.

First, let's define the names.

File   Edit   View

```
        moon: [
            {
                roles: ["colonyDome", "mine", "prison"],
                names: ["Fairfield (Colony Dome)", "New Haven (Mine)", "Pentridge (Prison)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "planetFall2_mainSurface_distributionCentre"],
                names: ["Penrith (Capital City)", "Kingswood (City)", "Boni Beach (Leisure Centre)", "Paddington Markets (Distribution Centre)"
            },
        ],
    };
}

this.$namingCricketStadiums = function(dataKey) {
    var names = ["Accor Stadium", "Lord's Cricket Ground", "Melbourne Cricket Ground", "Edgbaston Cricket Ground", "Sydney Cricket Ground"];
}
```

Ln 47, Col 1   141 of 1,784 characters   100%   Windows (CRLF)   UTF-8

Next, we need to pick one at random.

```
                roles: ["colonyDome", "mine", "prison"],
                names: ["Fairfield (Colony Dome)", "New Haven (Mine)", "Pentridge (Prison)"],
            },
        ],
    };
    pf._locationOverrides["0 62"] = {
        main: [
            {
                roles: ["capitalCity", "capitalCity", "leisureComplex", "planetFall2_mainSurface_distributionCentre"],
                names: ["Penrith (Capital City)", "Kingswood (City)", "Boni Beach (Leisure Centre)", "Paddington Markets (Distribution Centre)"
            },
        ],
    };
}

this.$namingCricketStadiums = function(dataKey) {
    var names = ["Accor Stadium", "Lord's Cricket Ground", "Melbourne Cricket Ground", "Edgbaston Cricket Ground", "Sydney Cricket Ground"];
    var selected = Math.floor(system.scrambledPseudoRandomNumber(1123) * 5);
    return names[selected];
}
```
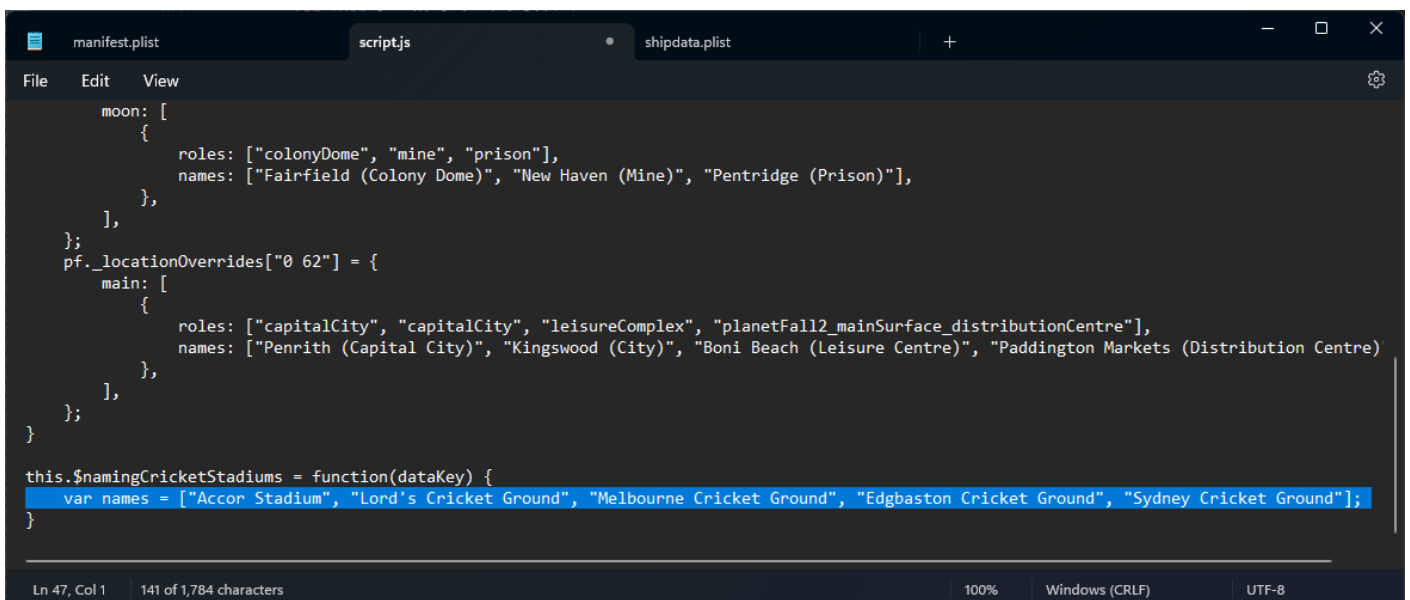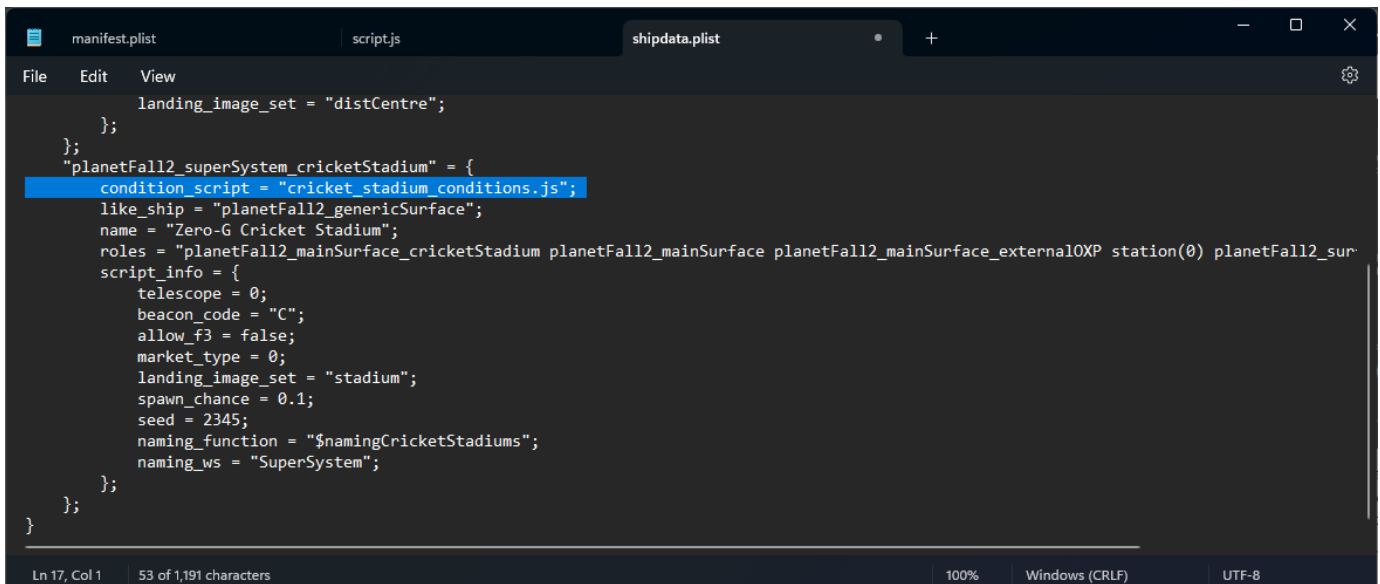
We're creating a variable called "selected", and doing a calculation, the result of which is stored in "selected". The calculation is: pick a system-defined random number between 0 and 1 (seeded with 1123). Whenever a random number is generated, it is almost always greater than or equal to 0 and less than 1 (ie. 0.99999999+more and below). Then multiply it by 5. That will create a random number greater than or equal to 0, and less than 5. Finally strip of any decimal values (which is what Math.floor does), so we are left with an integer between 0 and 4.

The last line of code uses the "selected" variable as the **index** for the "names" **array**, and returns the associated name.

At this point, our stadiums will be created fairly randomly but infrequently, and scattered anywhere around the galaxy. However, there are only 31 systems that are known to play Zero-G cricket. Maybe we should switch things up a bit and only have our new location spawn in those systems, but to always spawn (ie not randomly).

To achieve that, we need to go back to our shipdata.plist file and add a "condition_script" entry.

```
            landing_image_set = "distCentre";
        };
    };
    "planetFall2_superSystem_cricketStadium" = {
        condition_script = "cricket_stadium_conditions.js";
        like_ship = "planetFall2_genericSurface";
        name = "Zero-G Cricket Stadium";
        roles = "planetFall2_mainSurface_cricketStadium planetFall2_mainSurface planetFall2_mainSurface_externalOXP station(0) planetFall2_sur
        script_info = {
            telescope = 0;
            beacon_code = "C";
            allow_f3 = false;
            market_type = 0;
            landing_image_set = "stadium";
            spawn_chance = 0.1;
            seed = 2345;
            naming_function = "$namingCricketStadiums";
            naming_ws = "SuperSystem";
        };
    };
}
```

That extra line points to a script file, which we now need to create. We need two things: the "Scripts" folder, and "cricket_stadium_conditions.js".
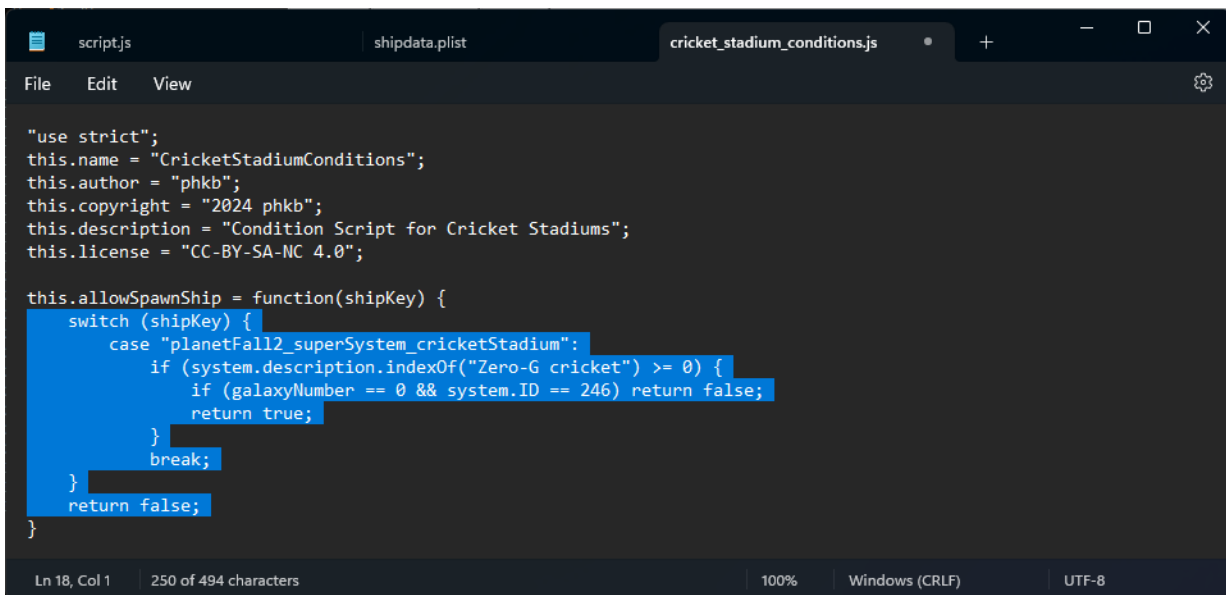
Condition scripts
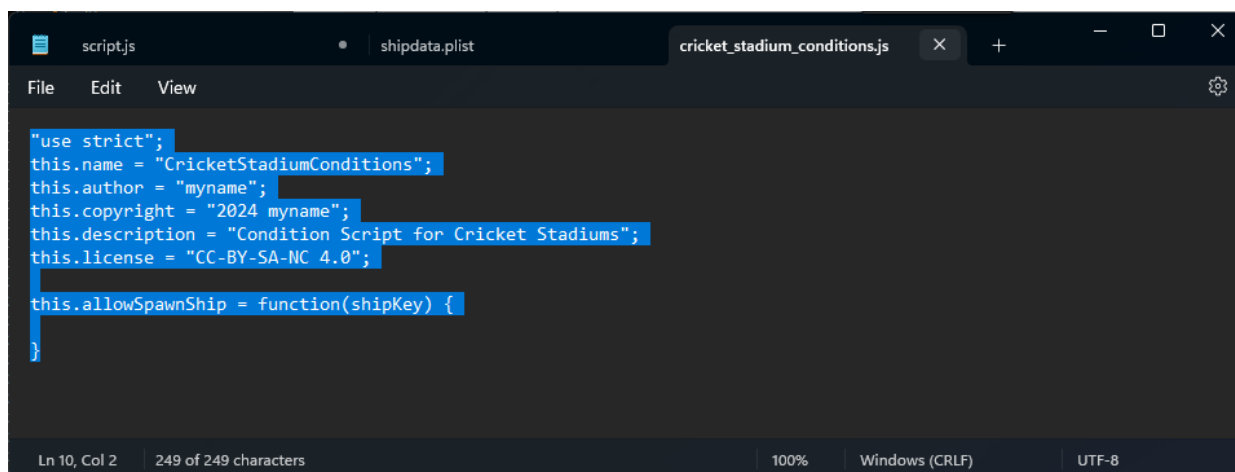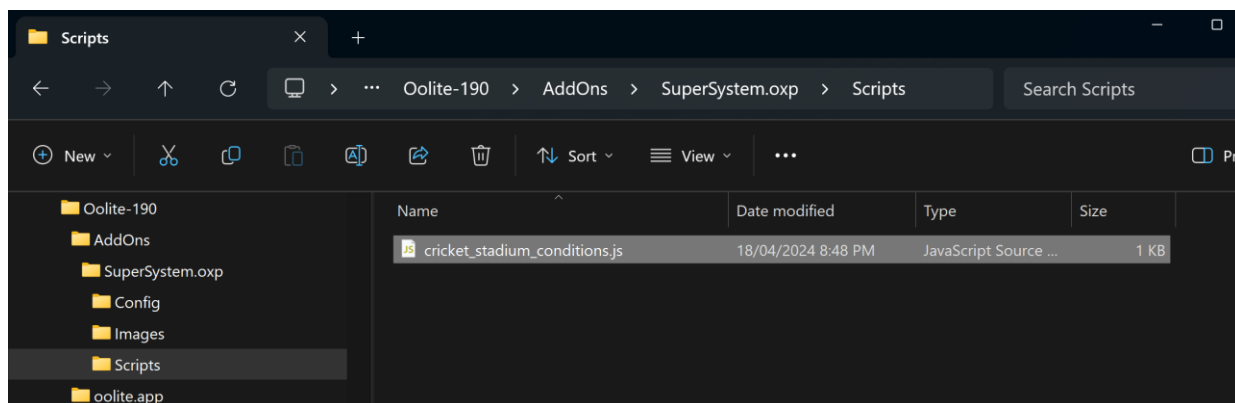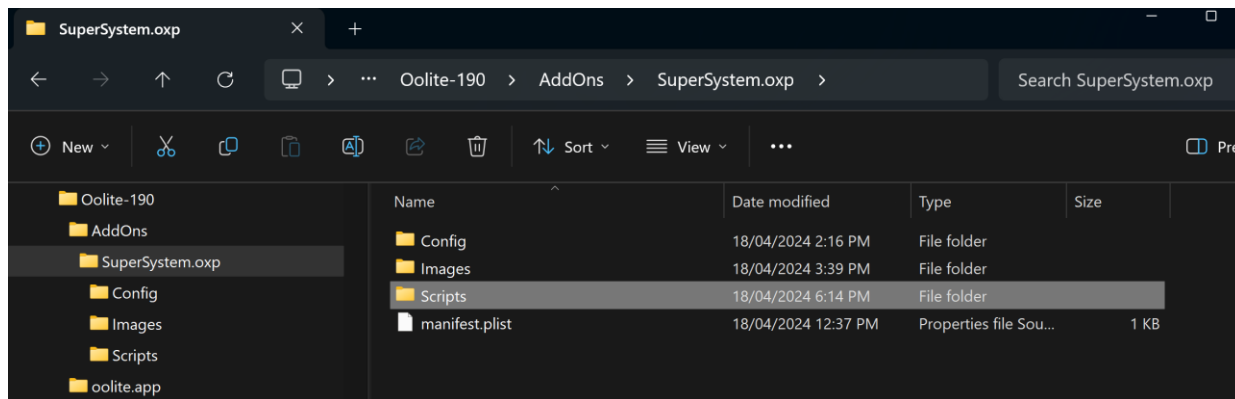
The format of this script file is similar to our world script, except the event we're hooking into is "allowSpawnShip". A "shipKey" parameter is being passed to this function, which is the key from the shipdata.plist file. In this sample project, there would only be one value ever passed through, that being "planetFall2_superSystem_cricketStadium", but to help with potential expansions of this concept, let's work on it as if multiple ships might be using it.

So, we need to return either a "true" value from this function, meaning, yes, the ship is OK to spawn, or a "no" value, meaning the ship cannot be spawned. The easiest way to do this is by using a "switch" statement. It kind of looks like this:

```
switch (shipKey) {
    case "planetFall2_superSystem_cricketStadium":
        if (system.description.indexOf("Zero-G cricket") >= 0) {
            if (galaxyNumber == 0 && system.ID == 246) return false;
            return true;
        }
        break;
}
return false;
```

File   Edit   View

```js
"use strict";
this.name = "CricketStadiumConditions";
this.author = "phkb";
this.copyright = "2024 phkb";
this.description = "Condition Script for Cricket Stadiums";
this.license = "CC-BY-SA-NC 4.0";

this.allowSpawnShip = function(shipKey) {
    switch (shipKey) {
        case "planetFall2_superSystem_cricketStadium":
            if (system.description.indexOf("Zero-G cricket") >= 0) {
                if (galaxyNumber == 0 && system.ID == 246) return false;
                return true;
            }
            break;
    }
    return false;
}
```

Ln 18, Col 1   250 of 494 characters   100%   Windows (CRLF)   UTF-8

So what's going on here? The "switch" statement is checking the value of shipKey against various "cases". The first (and admittedly, only) case is for "planetFall2_superSystem_cricketStadium". If shipKey is equal to "planetFall2_superSystem_cricketStadium", the next bit of code will run.

The next bit is then checking the description property of the current system object. The system object has things like the system ID, government type, economic type, a load of various properties that make up a system, and one of those properties is the description. It is a text string, that looks kind of like this:

```
"The planet Zadies is famous for its inhabitants' exceptional love for food blenders but scourged
by dreadful solar activity."
```

The important thing about the description, though, is it tells us who likes Zero-G cricket, and therefore who is likely to have a stadium. So, we are using the "indexOf" method of the string object to search for any instance of "Zero-G cricket" in the description. If it finds one, the returned value will be a number greater than or equal to zero. If it doesn't find it, the returned value is -1. Thus, we're looking for a value greater than or equal to zero.

But there is a catch. For one system, they actually loathe Zero-G cricket. That system is Tianve (ID 246 in galaxy number 0). So, after we have confirmed that Zero-G cricket is in the description, we also need to check for this edge case. If we're in Tianve, we definitely do not want a Zero-G stadium. So, we would return false in that case.

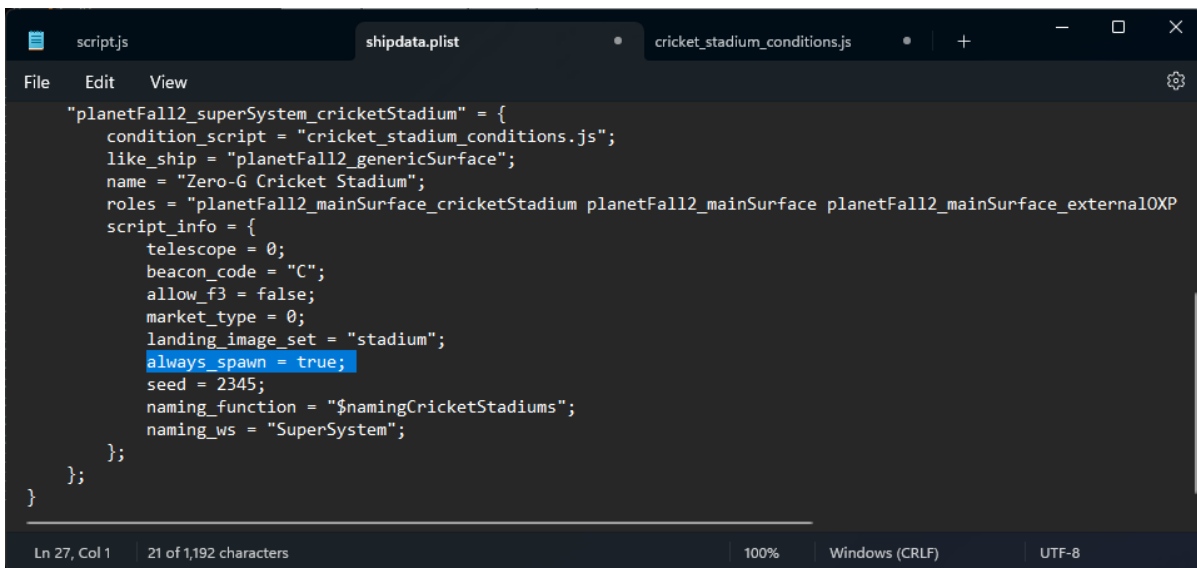But for any other system with Zero-G cricket, we would return true.

At this point, any other shipKey would fail to find a matching "case", and the code would end up returning false.

The final change we need to make is to the shipdata.plist entry for our stadium.

You might remember that we previous set up our entity with a spawn_chance of 0.1. We could change that to "1", which would work. But we can also do this:

```
always_spawn = true;
```

```
  "planetFall2_superSystem_cricketStadium" = {
      condition_script = "cricket_stadium_conditions.js";
      like_ship = "planetFall2_genericSurface";
      name = "Zero-G Cricket Stadium";
      roles = "planetFall2_mainSurface_cricketStadium planetFall2_mainSurface planetFall2_mainSurface_externalOXP
      script_info = {
          telescope = 0;
          beacon_code = "C";
          allow_f3 = false;
          market_type = 0;
          landing_image_set = "stadium";
          always_spawn = true;
          seed = 2345;
          naming_function = "$namingCricketStadiums";
          naming_ws = "SuperSystem";
      };
  };
}
```

What this will do is not even try to do the random chance calculation, and just always try to spawn this item. Without the brake implemented via our condition script, that would mean every system would end up with a Zero-G stadium. But instead, what will happen is that in every system PlanetFall will *try* to spawn it, but the condition script will cause it to fail, except in those systems we want it to.

Downloads

There is a download for all the code in this sample:

SuperSystem.oxp.zip