

## 1 Introduction

The scripting facility is Stellarium’s version of a *Presentation*, a feature that may be used to run an astronomical or other show for instruction or entertainment from within the Stellarium program. The use of scripts was recognized as a perfect way of arranging a presentation of a sequence of astronomical events from the earliest versions of Stellarium.

The original *Stratoscript* was quite limited in what it could do, and so a new Stellarium Scripting System has been developed.

Since version 0.10.1, Stellarium has included a scripting feature based on the Qt5 Scripting Engine<sup>1</sup>. This made it possible to write small programs within Stellarium to produce automatic presentations, set up custom configurations, and to automate repetitive tasks.

By version 0.14.0 a new scripting engine had reached a level where it had all required features for usage, and support of scripts for the old *Stratoscript* engine has been discontinued.

The programming language ECMAScript<sup>2</sup> (also known as JavaScript) gives users access to all basic ECMAScript language features such as flow control, variables, string manipulation and so on. Its integration with Qt’s QtScript module and the way Stellarium’s main components (“StelModule”s) have been designed to work means that all module functions which are labeled as “slots” can be called from JavaScript.

Interaction with Stellarium-specific features is done via a collection of objects which represent components of Stellarium itself. The various modules of Stellarium, and also activated plugins, can be called in scripts to calculate, move the scene, switch on and off display of objects, etc. You can write text output into text files with the `output()` command. You can call all public slots which are documented in the scripting API documentation<sup>3</sup>.

With the adaptation to Qt6 in versions 1.0 and later we had to switch to yet another scripting engine in 2022. Qt6 comes with QJSEngine, another JavaScript engine which behaves mostly similar to the older QtScript. However, small differences exist, so that care must be taken if scripts should be developed for both series of Stellarium. Some older scripts will not work on versions 1.0 and later or at least require some adaptations described in section 5.

## 2 The Script Console

It is possible to load, edit, run and save scripts using the script console window. To toggle the script console, press F12.

### 2.1 The Tabs in the Console

The three tabs “Script”, “Log” and “Output” each provide a text field. In the “Script” text field you can edit a script, which can be loaded from and saved to a file, and executed. The “Log” text field receives all errors and other messages resulting from the script’s execution; the “Output” text field will show the results of `core.output()` function calls.

The fourth tab, “Settings”, provides some configuration options.

### 2.2 The Menu Bar

The menu bar contains buttons for loading and saving a script. A file dialog is shown for selecting the file. The “clear” button lets you clear the contents of the text field in the currently selected tab. The drop-down menu “Execute:” offers a selection of menu entries, mainly intended to let you restore the sky to a clean state. The entry “selected text as script” executes selected text in the “Script” text field, allowing to test scripts during development. Entries “remove screen text”, “remove screen images” and “remove screen markers” remove labels, images and markers, respectively, that may have been left over on the screen from previous script actions. Each of the “clear

<sup>1</sup><https://doc.qt.io/qt-5/qtscript-index.html>

<sup>2</sup><https://en.wikipedia.org/wiki/ECMAScript>

<sup>3</sup> <https://www.stellarium.org/doc//scripting.html>

map” entries results in a call to `core.clear()`, using one of the arguments “natural”, “starchart”, “deepspace”, “galactic” or “supergalactic”. These calls reset Stellarium’s state, resulting in a basic state of the display and providing a clean starting point for another run of your script.

The button SSC runs the Stellarium Script Preprocessor with the script in the text field as input and produces an output by recursively inserting all included scripts into the text. *This output replaces the contents of the Script text field. Store the original script if you need it!*

## 2.3 German Keyboards

There is a glitch in the keyboard module of the underlying system that is responsible for keyboard handling. It affects users of Windows and Linux systems with a German, and probably also other international keyboards: The key combinations AltGr+8 and AltGr+9 cannot be used for typing [ and ]. You need to work around by copy-pasting these two characters. You can also try to use a command line argument like `-platform windows:altgr`.

## 3 Includes

Stellarium provides a mechanism for splitting scripts into different files. Typical functions or lists of variables or celestial objects can be stored in separate .inc files and used within other scripts through the `include()` command: `include("common_objects.inc");`

## 4 Minimal Scripts

This script prints “Hello Universe” in the Script Console log window and into log.txt: `core.debug("Hello Universe");`

This script prints “Hello Universe” in the Script Console output window and into the file output.txt which you will also find in the user data directory. The absolute path to the file will also be given in log.txt. `core.output("Hello Universe");` The file output.txt will be rewritten on each run of Stellarium. In case you need to save a copy of the current output file to another file, call `core.saveOutputAs("myImportantData.txt");` `core.resetOutput();`

This script uses the LabelMgr module to display “Hello Universe” in red, fontsize 20, on the screen for 3 seconds. `var label=LabelMgr.labelScreen("Hello Universe", 200, 200, true, 20, "ff0000");` `core.wait(3);` `LabelMgr.deleteLabel(label);`

## 5 Critical Scripting Differences introduced with version 1.0

Qt5’s QtScript module which has been used in versions 0.10 to 0.22 was more flexible and easier to fine-tune than Qt6’s QJSEngine, which is used in the 1.\* series of the program, i.e., versions 1.0 (22.3) and later. Still, most scripts that have been developed for Stellarium versions 0.10 to 0.22 still run without or with just minor modifications as described in this section.

### 5.1 Pause/Resume

The Qt6-based builds (version 1.0 and later) do not offer a way to pause and resume a script as was available in the Qt5-based builds (0.10 to 0.22).

### 5.2 The Vec3f problem

One of the most important little helper classes in the main program are three-dimensional vectors. `Vec3f` describe those with single-precision floating point data, and `Vec3d` are those with double-precision floating point data.

Input	Result with Qt5
x var f1=new V3d(.1,.2,.3);	
core.output(f1);	[0.1, 0.2, 0.3]
core.output(+f1.toString());	[0.1, 0.2, 0.3]
x core.output(f1.toHex());	#19334c
x core.output(f1.toVec3d());	[0.1, 0.2, 0.3]
x var f2=new V3d(f1.x(), 2*f1.y(), 3*f1.z());	
core.output(f2);	[0.1, 0.4, 0.9]
x core.output(f2.toVec3d());	[0.1, 0.4, 0.8999999999999999]
x var crimson=new Color("Crimson");	
x core.output(crimson.toHex());	#dc143c
core.output(crimson.toVec3f());	[r:0.8627451062202454, g:0.0784313753247261, b:0.23529411852359772]
core.output(crimson.r);	0.8627451062202454
core.output(crimson.r());	error
x core.output(crimson.getR());	0.8627451062202454
x core.output(crimson.getG());	0.0784313753247261
x core.output(crimson.getB());	0.23529411852359772
crimson.g=0.444;	(works)
crimson.setG(0.444);	(error)
x crimson=new Color(crimson.getR(), 0.444, crimson.getB());	(reassign crimson; recommended for Qt5 and Qt6)
x core.output(crimson.toRGBString());	[r:0.862745, g:0.444, b:0.235294]
x var eq=new Color(GridLinesMgr.getColorEquatorJ2000Grid());	
x core.output(eq.toHex());	#00aaff
core.output(eq.toVec3f());	[r:0, g:0.6666669845581055, b:1]
core.output(eq.toVec3d());	(n.a.)
core.output(eq.toString());	[0, 0.666667, 1]
x core.output(eq.toRGBString());	[r:0, g:0.666667, b:1]
x var c=new Color(core.vec3f(.3, .4, .5));	
core.output(c.toString());	[0.3, 0.4, 0.5]
core.output(c.toVec3f());	[r:0.30000001192092896, g:0.4000000059604645, b:0.5]
x var d=c; // assign to other	
core.output(d.toVec3f());	[r:0.30000001192092896, g:0.4000000059604645, b:0.5]
x var c2=new Color(.3, .4, .5);	
core.output(c2.toString());	[0.3, 0.4, 0.5]
core.output(c2.toVec3f());	[r:0.30000001192092896, g:0.4000000059604645, b:0.5]
x var d2=c2; // assign to other	
core.output(d2.toVec3f());	[r:0.30000001192092896, g:0.4000000059604645, b:0.5]

Use of V3d, V3f and Color wrapper classes. Only use the calls marked with x in scripts targeted at all versions of Stellarium.

As late as in version 0.19 we added **Vec3f** and later **Vec3d** data types to the Javascript environment. These allowed addressing variables of C++ classes **Vec3f** or **Vec3d** with named sub-fields **r**, **g**, **b** (if used to describe colors), or **x**, **y**, **z** (same data but contextually understood as 3D data). These functions may have pleased the JavaScript connoisseur when running scripts with Qt5-based builds of Stellarium (series 0.\*, now usually found on 32-bit systems), but we recommend to abstain from their use in new scripts.

Qt6's scripting module cannot be extended that easily. The **Vec3d** and **Vec3f** data types are not directly scriptable, but we can deal with method arguments of these types when they are mentioned in the API documentation (see 1). To access internals of these classes, we must use wrapper classes which bear different names:

**V3d** can be used where **Vec3d** must be accessed.

**V3f** can be used where **Vec3f** must be accessed.

**Color** can be used where a **Vec3f** must be accessed which represents a color value.

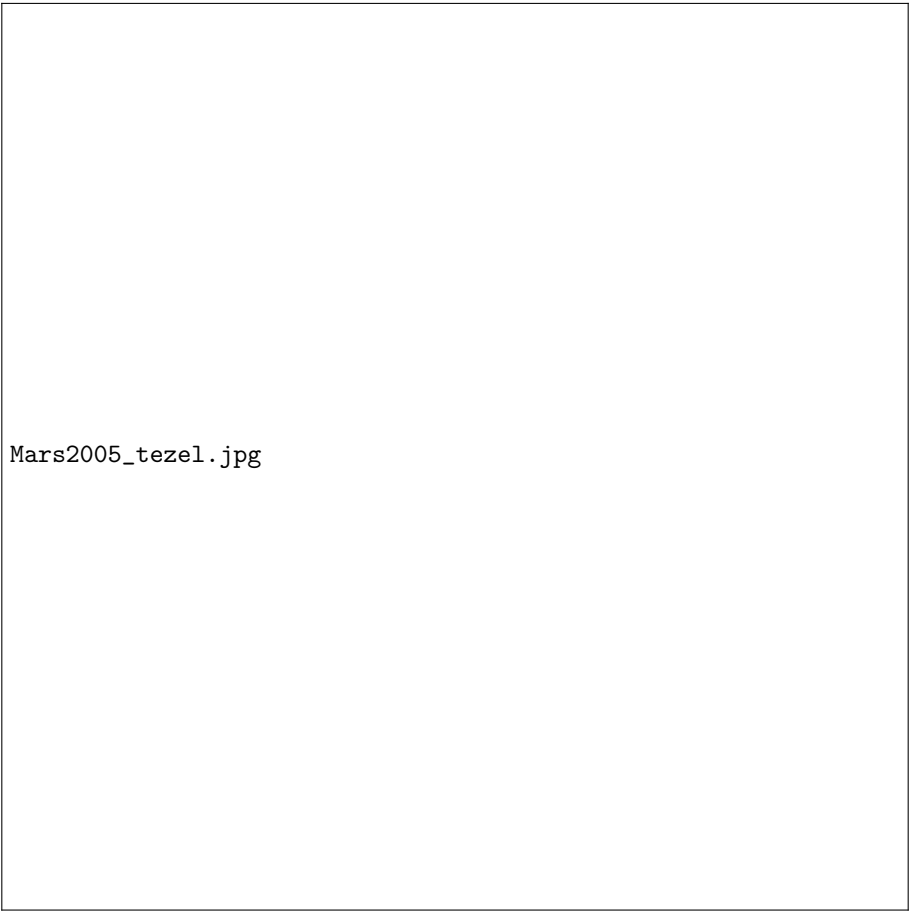
They behave slightly different when run using Qt5 or Qt6, as shown in Table 5.2. If you aim for maximum compatibility between versions, only use the calls which provide equal results in both series. This current state is far from optimal or even pretty, but a pragmatic solution that works. We invite advanced developers to find a better solution that works with both Qt5 and Qt6.

## 6 Example: Retrograde motion of Mars

A good way begin writing of scripts: set yourself a specific goal and try to achieve it with the help of few simple steps. Any complex script can be split into simple parts or tasks, which may solve any newbie problems in scripting.

Let me explain it with examples.

Imagine that you have set a goal to make a demonstration of a very beautiful, but longish phenomenon — the retrograde motion of the planet Mars (Fig. 1).



Mars2005\_tezel.jpg

Figure 1: Retrograde motion of Mars in 2005. (Credit & Copyright: Tunc Tezel — APOD: 2006 April 22 – Z is for Mars.)

## 6.1 Script header

Any “complex” script should contain a few lines in the first part of the file, which contains important data for humans — the name of the script and its description — and some rules for Stellarium. You can even assign default shortcuts in the script header, but make sure you assign a key not used elsewhere! The shortcuts are read during startup from all scripts in the scripts sub-directories of both program and user data directories (see section ??). The description may cover several lines (until the end of the commented header) and should therefore be the last entry of the header.

```
// // Name: Retrograde motion of Mars // Author: John Doe // License: Public Domain //  
Version: 1.0 // Shortcut: Ctrl+M // Description: A demo of retrograde motion of Mars. //
```

## 6.2 A body of script

At the first stage of writing of the script for a demo of retrograde motion of Mars we should set some limits for our demo. For example we want to see motion of Mars every day during 250 days since October 1<sup>st</sup>, 2009. Choosing a value of field of view and of the coordinates of the center of the screen should be done at the this stage also.

Let’s add few lines of code into the script after the header and run it: `core.setDate("2009-10-01T10:00:00"); core.moveToRaDec("08h44m41s", "+18d09m13s", 1); StelMovementMgr.zoomTo(40, 1); for (i=0; i<250; i++) core.setDate("+ 1 days"); core.wait(0.2);`

OK, Stellarium is doing something, but what exactly is it doing? The ground and atmosphere is enabled and any motion of Mars is invisible. Let’s add an another few lines into the script (hiding the landscape and atmosphere) after setting date and time:

```
LandscapeMgr.setFlagLandscape(false); LandscapeMgr.setFlagAtmosphere(false);
```

The whole sky is moving now — let’s lock it! Add this line after previous lines: `StelMovementMgr.setFlagLockEquPos(true);`

It looks better now, but what about cardinal points, elements of GUI and some “glitch of movement”? Let’s change the script: `core.setDate("2009-10-01T10:00:00"); LandscapeMgr.setFlagCardinalsPoints(false); LandscapeMgr.setFlagLandscape(false); LandscapeMgr.setFlagAtmosphere(false); core.setGuiVisible(false); core.moveToRaDec("08h44m41s", "+18d09m13s", 1); StelMovementMgr.setFlagLockEquPos(true); StelMovementMgr.zoomTo(40, 1); core.wait(2); for (i=0; i<250; i++) core.setDate("+ 1 days"); core.wait(0.2); core.setGuiVisible(true);`

It’s better, but let’s draw the “path” of Mars! Add those line before the loop: `core.selectObjectByName("Mars", false); SolarSystem.setFlagIsolatedTrails(true); SolarSystem.setFlagTrails(true);`

Hmm...let’s add a few strings with info for users (insert those lines after the header): `var color = "ff9900"; var info = LabelMgr.labelScreen("A motion of Mars", 20, 20, false, 24, color); var apx = LabelMgr.labelScreen("Setup best viewing angle, FOV and date/time.", 20, 50, false, 18, color); LabelMgr.setLabelShow(info, true); LabelMgr.setLabelShow(apx, true); core.wait(2); LabelMgr.setLabelShow(apx, false);`

Let’s add some improvements to display info for users — change in the loop: `var label = LabelMgr.labelObject(" Normal motion, West to East", "Mars", true, 16, color, "SE"); for (i=0; i<250; i++) core.setDate("+ 1 days"); if ((i var strDate = "Day " + i; LabelMgr.setLabelShow(apx, false); var apx = LabelMgr.labelScreen(strDate, 20, 50, false, 16, color); LabelMgr.setLabelShow(apx, true); if (i == 75) LabelMgr.deleteLabel(label); label = LabelMgr.labelObject(" Retrograde or opposite motion begins", "Mars", true, 16, color, "SE"); core.wait(2); LabelMgr.deleteLabel(label); label = LabelMgr.labelObject(" Retrograde motion", "Mars", true, 16, color, "SE"); if (i == 160) LabelMgr.deleteLabel(label); label = LabelMgr.labelObject(" Normal motion returns", "Mars", true, 16, color, "SE"); core.wait(2); LabelMgr.deleteLabel(label); label = LabelMgr.labelObject(" Normal motion", "Mars", true, 16, color, "SE"); core.wait(0.2);`

## 7 More Examples

The best source of examples is the scripts sub-directory of the main Stellarium source tree. This directory contains a sub-directory called tests which are not installed with Stellarium, but are nonetheless useful sources of example code for various scripting features.