MARI

SOFTWARE API OVERVIEW

VERSION 2.6V3

THE FOUNDRY.

Mari software engineering: Jack Greasley, Kiyoyuki Nakagaki, Marcus Shoo, Kevin Atkinson, Tim Ebling, Jed Soane, Daniel Lond, Robert Fanner, Duncan Hopkins, Mark Final, Chris Bevan, Carl Rand, Phil Hunter, Tim Smith, and Rajiv Perseedoss

Product testing: Michael Zannetou, Mark Titchener, Robert Elphick, Antoni Kujawa, Ravishankar Raju, Chris Hiess, Jorel Latraille, and John Crowe

Writing and layout design: Jack Elder, Jon Hertzig, Eija Närvänen, Charles Quinn, and Erica Cargle

Proof reading: Jack Elder, Eija Närvänen, Joel Byrne, Charles Quinn, Erica Cargle, and Simon Picard

Mari includes Disney technology licensed from Walt Disney Animation Studios.

The Foundry

5 Golden Square,

London, W1F 9HT

Rev: Friday, August 15, 2014

CONTENTS

1 Preface

Mari is a creative texture-painting tool that can handle extremely complex or texture heavy projects. It was developed at Weta Digital and has been used on films such as The Adventures of Tintin: The Secret of the Unicorn, District 9, The Day the Earth Stood Still, The Lovely Bones, and Avatar.

The name Mari comes from the Swahili 'Maridadi', meaning 'beautiful' and carrying connotations of 'usefulness'.

About this Guide

This guide is aimed at developers and provides you with the basic information you need to get started using the Python, Shader and C APIs in Mari.

For further information on Mari and its functions, see the accompanying *Mari User Guide* and *Mari Reference Guide*.

Contact Customer Support

Should questions arise that this guide fails to address, you can contact Customer Support directly via e-mail at support@thefoundry.co.uk or via telephone to our London office on +44 (0)20 7479 4350 or to our Los Angeles office on (310) 399-4555 during office hours.

2 Using Python in Mari

Introduction

Mari uses Python 2.6.5 and Unicode Character Set (UCS) 4.

PythonQt provides access to almost all of the Qt libraries (Qt is a C++ GUI library developed by Qt Development Frameworks). If you've used PyQt in the past, PythonQt looks very similar:

```
# PyQt version
from PyQt4 import QtGui
# PythonQt version
from PythonQt import QtGui
# Everything else is the same in this example
w = QtGui.QLabel("hello")
w.show()
```

For more information on Qt and PythonQt, visit http://qt.nokia.com/ and http://pythonqt.sourceforge.net/.

Most of Mari's functions are implemented through the use of manager objects, such as **mari.projects**, **mari.menus**, and so on.

Steps for Using Python in Mari

To use Python in Mari, follow these steps:

1. Review Mari's Python Documentation
2. Enter Python Statements in the Python Console Palette
3. If you want your script to automatically run on start-up, Save or Import a Script to:
   - **~/Mari/Scripts**, or
   - your own directory by setting the environment variable **MARI_SCRIPT_PATH** to a custom folder.
4. Load a Script via the **Script Path** entry box.
5. Use the Terminal Mode from a shell.
6. Using PySide in Mari

Review Mari's Python Documentation

1. To view HTML documentation on Mari's Python API, select **Python** > **API** in Mari.
2. To view example Python scripts, go to the **Media/Scripts/examples** sub-directory of the Mari application directory and open any of the **.py** files there in a text editor. You can also see the results of these example scripts in the **Python** menu in Mari.

NOTE: Executing the Python "help" command in the **Python Console** also launches the HTML documentation in a web browser.

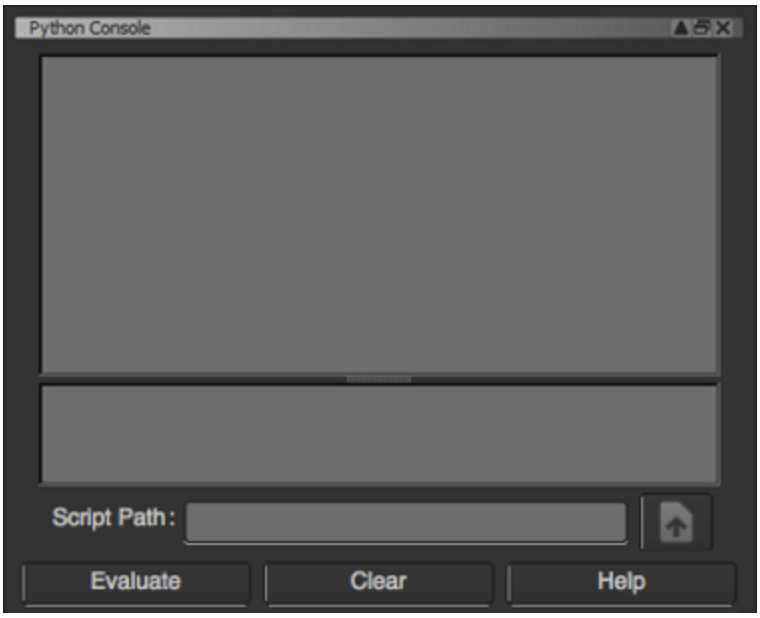
TIP: To read more about Python, review its documentation, or interact with other Python users, you can also visit the Python programming language official website at http://www.python.org/.

Enter Python Statements in the Python Console Palette

Mari's **Python Console** palette behaves in a very similar way to a standard Python console. Here's how you use it:

1. If the **Python Console** is already open, click the **Console** tab to give it focus; or if it's closed, right-click in the toolbar area on top of the Mari workspace and select **Python Console** to open it. You can also find it in the menu under **View** > **Palettes**.

   The **Python Console** is divided into two parts. You use the lower part (input pane) to type in and execute your Python statements, and when you have done so, statements and their outputs appear in the upper part of the console (output pane).
2. To enter a statement, type it into the input pane. As you type, Mari provides auto-complete suggestions, if any are available.



To use the usual editing functions, such as copy and paste, right-click on the input pane and select the desired function.

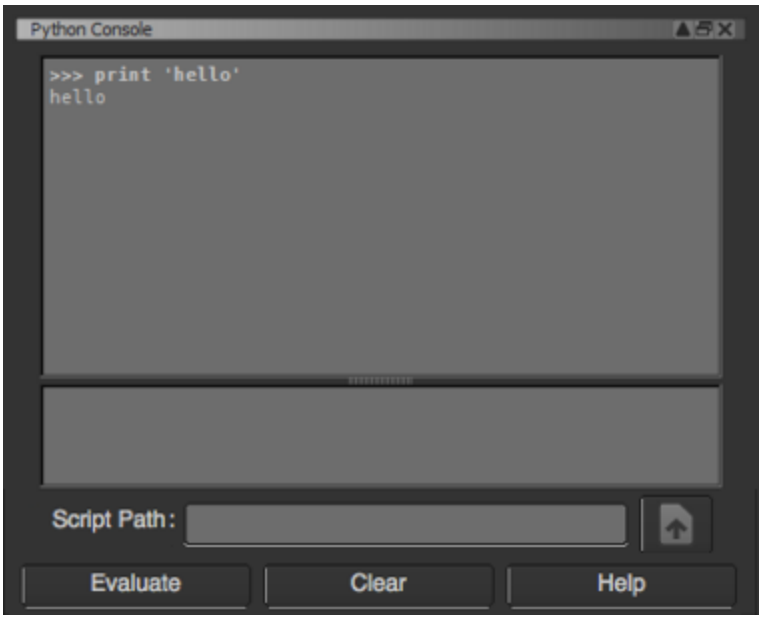
TIP: To temporarily turn off Mari's auto-complete function while writing in the Python Console palette, press **Esc** when the auto-complete word appears. This only causes it to disappear until you continue typing, however, and auto-complete appears again if it detects a similar word when you resume.

3. Mari utilizes a syntax highlighting scheme in the input pane, so that code can be more easily identified at a glance. Text in red represents strings, dark blue represents comments, and various Python keywords are represented by yellow, magenta, and blue.

```
for n in range (0,10) :
# comment code here
if blah:
   print "yay"
   else
   mari.canvases.paintBuffer().bake()
```

4. To execute the statement, click the **Evaluate** button or press **Ctrl**/**Ctrl**+**Cmd**+**Return.**

   Your statement disappears from the input pane, and appears in the output pane, preceded by **>>>** (or **...** if your statement spans multiple lines).



If you enter an invalid statement, Mari produces an error in the output pane.

5. If you want to repeat a statement, you can step backwards or forwards through the history of your script by pressing **Ctrl**/**Ctrl**+**Cmd**+**Up** or **Ctrl**/**Ctrl**+**Cmd**+**Down**.
6. If you want to clear the input and output panes, click the **Clear** button.

NOTE: Sometimes you may get an error if you copy and paste statements into the **Python Console** from another source, like an e-mail. This may be caused by the mark-up or encoding of the source you copied the statement from. To fix the problem, re-enter the statement manually.

TIP: You can increase or decrease the font size in the **Python Console** by holding down **Ctrl**/**Ctrl**+**Cmd** and pressing **+** or **-**.

7. If you want to quickly open the **Mari Python API Help** without going to the **Python** > **Documentation** menu, click on **Help** in the **Python Console** palette.

   A window with the **Mari Python API Help** appears. You can move this window around and keep typing in the input pane while it's open. This window cannot be docked like a palette, but you can change focus from the window, while it's still active, unlike most of Mari's dialogs.

NOTE: If you are typing in the input pane while the **Mari Python API Help** window is open, and Mari detects an auto-complete suggestion, the suggestion appears in the **Mari Python API Help** automatically.

Save or Import a Script

If you would like your script to be automatically executed at start-up, do the following:

1. Enter your script in any text editor.
2. Save the text file with the extension **.py** (for example, **my_module.py**) to:

   • the **~/Mari/Scripts** sub-directory of the Mari application directory, or

   • your own directory by setting the environment variable **MARI_SCRIPT_PATH** to a custom folder.

   If a script called **init.py** exists, it is run first. If you need scripts to run in a specified order, you should put them in a separate folder and use a Python **import** statement in **init.py**.

   You can also specify multiple directories to run scripts on start-up, using a path list separated by '**:**' or '**;**' as per the standard for your operating system.

   • **On Linux**: /home/username/dir1:/home/username/dir2

   • **On Windows**: C:\Users\username\dir1;C:\Users\username\dir2

   • **On Mac:** /home/users/username/dir1;/home/users/username/dir2
3. The **import** directory, contained in the following folders of your custom scripts path:

   • **On Linux:** /Mari/Scripts

   • **On Windows:** Documents\Mari\Scripts.

   • **On Mac:** /Documents/Mari/Scripts

   Scripts saved to this directory are imported rather than run, which prevents unnecessary pollution of the global namespace. If you save your custom script to the **import** directory, it is automatically imported before the scripts in the main folder are run.

TIP: Python files and sub-folders that you wish to import without running in the **import** directory must be **__init__.py** files.

Load a Script

To load a script via the **Script Path** entry box:

1. Manually enter the script's location, or
2. Click on  to launch the **Python Script Path** dialog box and browse to the location of the script.
3. To execute the statement, click the **Evaluate** button.

Custom scripts load from the **import** directory prior to the scripts being run from the main folder. Mari uses particular order to load modules to ensure the proper, full functionality of the program when using Python scripts.

If existing start-up scripts do not work correctly due to the load module order, you can revert back to the old behavior by setting the environment variable **MARI_OLD_PYTHON_INIT** to any non-empty string other than 0.

NOTE: Mari looks for metadata that is over 250MB in size and discards anything over this size. This check is performed on project load, and is intended to strip corrupt and problematic data. This affects metadata added via the Python API.

Terminal Mode

Mari provides a terminal mode, which allows users to access features through a Python shell on the command line. This is similar to the Python console palette in the Mari GUI, or to a standard Python shell. Terminal mode can be used to perform administrative actions such as archiving projects, exporting textures, and other tasks that do not require graphical interaction from the user.

NOTE: Terminal mode is not a true "headless mode" because it still requires graphics functionality to run. Mari still initializes its GUI when starting up terminal mode - it just won't display it.

You can supply the file name of a Python script to run on the command line, if desired. If supplied, Mari runs the specified script before providing the Python shell input prompt. It is also possible to start in "execute mode", which is the same as terminal mode but exists after running the provided script.

To start the terminal mode from a shell, use the following commands:

On Linux

```
cd /path/to/mari
# Normal terminal mode
./mari -t
# Terminal mode - run the script, then show a Python input prompt
./mari -t /path/to/script_name.py
# Execute mode - run the script, then exit
./mari -x /path/to/script_name.py
```

On Windows

```
cd \path\to\mari
# Normal terminal mode
Mari2.6v3.exe -t
# Terminal mode - run the script, then show a Python input prompt
Mari2.6v3.exe -t \path\to\script_name.py
# Execute mode - run the script, then exit
Mari2.6v3.exe -x \path\to\script_name.py
```

On Mac

```
cd /path/to/Mari
# Normal terminal mode
./Mari2.6v3 -t
# Terminal mode - run the script, then show a Python input prompt
./Mari2.6v3 -t /path/to/script_name.py
# Execute mode - run the script, then exit
./Mari2.6v3 -x /path/to/script_name.py
```

Using PySide in Mari

Mari ships PySide together with its Python scripting engine and PythonQt. PySide is a library of Python bindings for Qt, similar to PyQt. For more information on PySide, visit http://qt-project.org/wiki/PySide.

Mari's Python bindings are based on PythonQt. There are a few points to note in using PythonQt-based bindings and PySide in conjunction with one another:

- It is possible to connect signals and slots between Mari's Python bindings and PySide using the **mari.utils.connect()** function.
- Some Mari functions return Qt types such as **QSize**, **QRect**, and **QPoint**. These are Qt types of PythonQt. If such objects needs passing to PySide, they require conversion. For example:

```
canvas_size = mari.app.canvasSize()
widget = PySide.QtGui.QWidget()
widget.resize(PySide.QtCore.QSize(canvas_size.width(),canvas_size.height())) #Cannot directly pass canvas_size
```

NOTE: As of the Mari 2.6v1 release, Mari uses a combination of PySide and PythonQt. Existing Python scripts may potentially need to be amended for them to continue working correctly.

WARNING: In future, PythonQt in Mari will be deprecated in favor of PySide. Please keep this in mind when creating Python scripts.

3 Using Mari's C API

Mari provides an SDK for developing binary plug-ins for functionality where Python scripting is not the best solution.

Introduction

The C API allows users to develop plug-ins to read and write custom image formats, or to read custom geometry formats. In these cases, passing data between the application and plug-ins can be done more efficiently by passing buffers of binary data rather than using intermediate Python types.

The choice to use C instead of C++ for the API provides better compatibility and simpler compilation for users, and as always, can still be used in plug-ins written in C++.

Review Mari's C API Documentation

For full details of the C API, including example plug-ins, please see the C API documentation under **Help** > **SDK** > **C API** > **Documentation**, or via the installation directory in the SDK folder.

4 Custom Shaders

Modular Shaders

Versions of Mari prior to the 2.0v1 release provided Python API hooks to register custom "modular shaders." However, the modular shader system has since been deprecated in favor of a more flexible node-graph-based system.

Custom Shaders and Layers

A custom shader API for Mari's node-graph-based system exists to give you the ability to write and register your own shaders. Mari renders a scene in the canvas using its extensible shader system, based on a node graph. Internally, within Mari, shaders and layers are in the form of nodes. The custom shaders and layers can then be added by supplying XML files to Mari, defining custom shaders and layers in the form of nodes. Once the node definition has been written in an **.xml** file, it can be registered to Mari using the Python API.

The Custom Shader API that is built into Mari can be accessed by navigating to **Help** > **Custom Shader API**. Additionally, you can read more on the Python and C API documentation by navigating to **Python** > **Documentation** and **Help** > **SDK** > **C API** > **Documentation** on the Mari menu bar.

If you have any questions or difficulties with writing or registering custom shaders, please contact support@thefoundry.co.uk for assistance.