MARI

SOFTWARE API OVERVIEW

VERSION 1.6v1

Software API Overview™. Copyright © 2012The Foundry Visionmongers Ltd. All Rights Reserved. Use of this user guide and the Mari software is subject to an End User License Agreement (the "EULA"), the terms of which are incorporated herein by reference. This user guide and the Mari software may be used or copied only in accordance with the terms of the EULA. This user guide, the Mari software and all intellectual property rights relating thereto are and shall remain the sole property of The Foundry Visionmongers Ltd. ("The Foundry") and/or The Foundry's licensors.

The EULA can be read in the Mari User Guide Appendices.

The Foundry assumes no responsibility or liability for any errors or inaccuracies that may appear in this user guide and this user guide is subject to change without notice. The content of this user guide is furnished for informational use only.

Except as permitted by the EULA, no part of this user guide may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of The Foundry. To the extent that the EULA authorizes the making of copies of this user guide, such copies shall be reproduced with all copyright, trademark and other proprietary rights notices included herein. The EULA expressly prohibits any action that could adversely affect the property rights of The Foundry and/or The Foundry's licensors, including, but not limited to, the removal of the following (or any other copyright, trademark or other proprietary rights notice included herein):

Mari™ software © 2012 The Foundry Visionmongers Ltd. All Rights Reserved.

Mari™ is a trademark of The Foundry Visionmongers Ltd.

In addition to those names set forth on this page, the names of other actual companies and products mentioned in this Getting Started Guide (including, but not the to, those set forth below) may be the trademarks or service marks, or registered trademarks or service marks, of their respective owners in the United States and/or other countries. No association with any company or product is intended or inferred by the mention of its name in this Software API Overview.

Linux ® is a registered trademark of Linus Torvalds.

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 and Phil Hunter
Product testing: Michael Zannetou, Mark Titchener, Robert Elphick and Antoni Kujawa
Writing and layout design: Jack Elder, Jon Hertzig, Eija Närvänen, Charles Quinn and Erica Cargle
Proof reading: Jack Elder and Eija Närvänen

Mari includes Disney technology licensed from Walt Disney Animation Studios.

The Foundry
6th Floor, The Communications Building,
48 Leicester Square,
London
WC2H 7LT

Rev: December 4, 2012

Contents

PREFACE

About this Manual 4
Contact Customer Support 4

USING PYTHON IN MARI

Introduction 5
Steps for Using Python in Mari 5
Review Mari's Python Documentation 5
Enter Python Statements in the Python Console Palette 6
Save or Import a Script 7
Load a Script 8
Terminal Mode 9
- On Linux 9
- On Windows 9

USING MARI'S C API

Introduction 10
Review Mari's C API Documentation 10

MODULAR SHADERS

Introduction 11
Examples of Mari Shader Modules 11
Dissection of a Shader Module 12
Loading Shader Modules with Python 15
Fixing GLSL Compilation Errors 15
Mipmap Behavior 16
Learning More 16

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 Manual

This manual is aimed at developers.

The first part of the manual provides you with the basic information you need to get started using Python in Mari.

The rest of the manual consists of detailed information on how to create and load custom shader modules using the OpenGL Shading Language 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 manual 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 7968 6828 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.

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** subdirectory 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/*.*

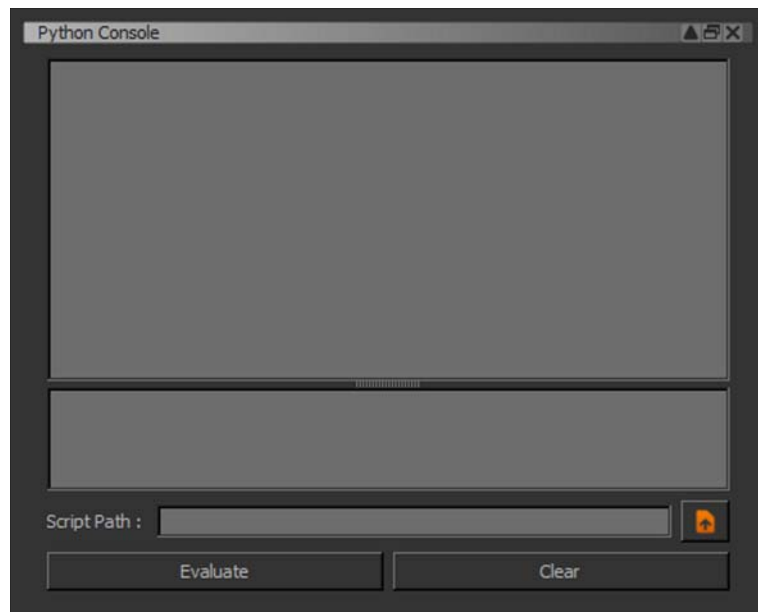
Enter Python Statements in the Python Console Palette

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).*

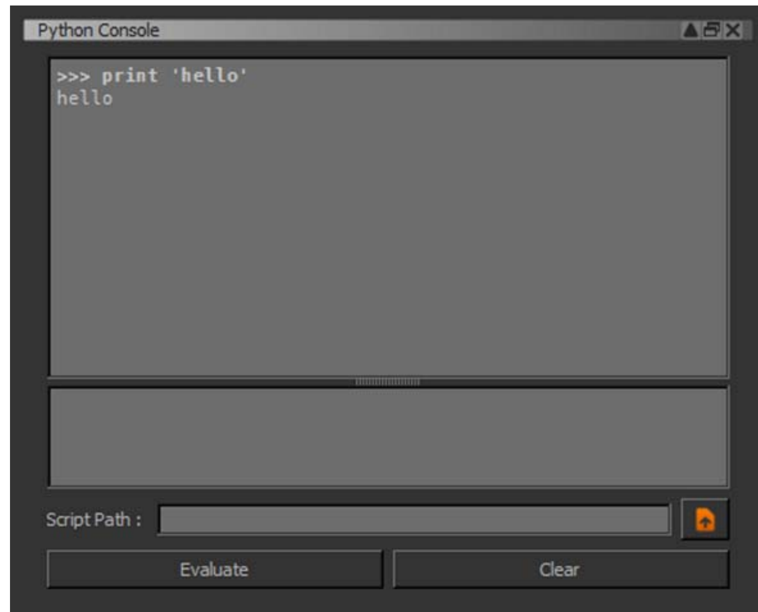
2. To enter a statement, type it into the input pane.



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

3. To execute the statement, click the **Evaluate** button or press **Ctrl+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.

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

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

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.*

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** subdirectory 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's ran 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*
3. The **Import** directory, contained in the following folders of your custom scripts path:
   - **On Linux:** /Mari/Scripts
   - **On Windows:** Documents\Mari\Scripts

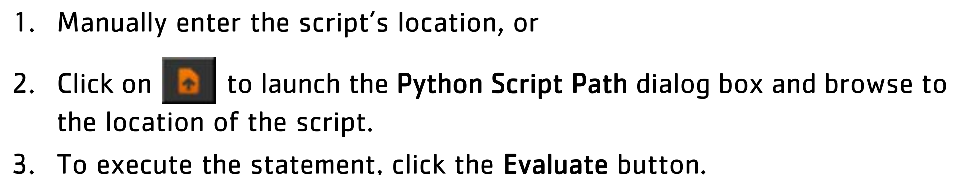
   import scripts without running them, which prevents unnecessary pollution of the global namespace. If you save your custom script to the **Import** directory, it is automatically imported before the scripts in the main folder are run.

Tip *Python files and subfolders 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 [icon] 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_PYTHNON_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
Mari1.4v4.exe -t
# Terminal mode - run the script, then show a Python input prompt
Mari1.4v4.exe -t \path\to\script_name.py
# Execute mode - run the script, then exit
Mari1.4v4.exe -x \path\to\script_name.py
```

3 Using Mari's C API

Introduction

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

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.

2 Modular Shaders

Introduction

Mari uses shaders for the real-time display of geometry. These shaders are made from a stack of modular components (modules), including a lighting module that is automatically added when a shader is created. Modules are executed in order, first to last, each one modifying the available shading parameters (diffuse, spec, bump, normal), always applying the lighting module last. Each module has a series of named parameters that can be adjusted. These parameters can be of the type **float**, **int**, **boolean** or **Channel**. **Channel** parameters can be set by providing the unique name of the channel as a string, or by passing a **Channel** instance.

Mari allows you to load custom shader modules that can be compiled for use and exposed in the UI. Mari's shader modules are written in the OpenGL Shading Language (GLSL), with some extended functions. Using GLSL you can add options to the **Add New Shader Module** dialog to create shaders matching production shaders or to add specific effects to existing shaders. Information embedded in comments in the GLSL shader files tells Mari how to add the files to the shader stack for a particular shader, and how to expose the modular shader's parameters with UI widgets (like sliders, for example).

To write your own custom shader, you need to embed the correct information in the source code comments, and use the extended functions to get, and set, the render state Mari provides.

Examples of Mari Shader Modules

Mari uses many shader files, with slightly differing information embedded in the comments, to achieve various functions. All of the shaders used in Mari are in the **Media/Shaders** subdirectory of the bundle directory. However, only the shader modules in **Media/Shaders/Modules** are relevant to our discussion. The shader modules in this directory are good examples of how to write your own custom shader modules for Mari.

Note *The files that are shipped with Mari in **Media/Shaders/Modules** should only be studied, and not modified. While Mari evaluates any new shader module files you place in this directory, the preferred method of adding custom shaders is through the Python API.*

Dissection of a Shader Module

Let's look at the code behind a typical shader module. This is the source code for the **diffuse** shader module, in the file **Sub_Diffuse.frag**, that ships with Mari.

```
//! begin_module    | Diffuse | Diffuse | Blends the input with the
underlying diffuse according to the Blend Amount.

//! include | Surface.glslh

//! module_call | sub_diffuse(State, Texture, BlendAmount);
//! module_declaration | void sub_diffuse(inout RenderState,
Channel, float);
//! input | Texture      | Texture      | Channel | channel |
diffuseColor
//! input | BlendAmount  | Blend Amount | float   | slider  | 1.0

void sub_diffuse(inout RenderState State, Channel texture, float
amount)
{
    vec4 blend  = texture_lookup(texture, State.UV);

    vec4 Result;
    vec4 base = State.diffuse;

    blend.a *= amount;

    blend = clamp( blend, 0.0, 1.0 );

    Result.a = 1.0 - ( 1.0 - blend.a ) * ( 1.0 - base.a );

    if(Result.a==0.0)
    {
        State.diffuse = base;
        return;
    }

    Result.rgb = ( base.a * ( 1.0 - blend.a ) * base.rgb + blend.a
* blend.rgb ) / Result.a;

    Result = clamp(Result, 0.0,1.0);

    State.diffuse = Result;
}
```

Let's dissect the code, starting from the top. Note how the first six lines all contain comments prefixed with '//!'. These comments provide special instructions to Mari.

The first comment tells Mari that the file constitutes a new shader module, with the internal name **Diffuse**, the UI name **Diffuse**, and a tooltip that reads: **Blends the input with the underlying diffuse according to Blend Amount.**

```
//! begin_module    | Diffuse | Diffuse | Blends the input with the
underlying diffuse according to the Blend Amount.
```

Next, we instruct Mari to include a core functionality shader file, **Surface.glslh**. This core file, and others that can be included in your module, are in **Media/Shaders/include**. It provides data, function declarations and definitions for working with Mari's renderstate. For example, it contains declarations for functions to look up texture information in Mari.

```
//! include | Surface.glslh
```

The following comment tells Mari how it should invoke the shader module function - it is invoked when combined with other shaders in a shader group. This line is inserted as provided in another GLSL file, along with any other shader modules, and the necessary glue code to make it all work.

```
//! module_call | sub_diffuse(State, Texture, BlendAmount);
```

The next line provides the function declaration to Mari.

```
//! module_declaration | void sub_diffuse(inout RenderState,
Channel, float);
```

Next, the input parameter and UI widget for each custom shader is specified, in the order they appear in the function argument list. The format is as follows:

```
internal name | UI name | c data type | UI widget | values for the
UI widget
```

```
//! input | Texture      | Texture      | Channel | channel |
diffuseColor
//! input | BlendAmount  | Blend Amount | float   | slider  | 1.0
```

Note

- ***Float** is an example of the `c data type` parameter.*
- *Some UI widgets can have extra comma separated values here, to indicate minimum, maximum and default values.*

In general, the values provided are the default values. The first parameter defaults to a channel named **diffuseColor**, and the second channel defaults to a float with a value of 1.0.

Additional UI examples can be found in the other shader module files that ship with Mari. For example, the **sub_bump.frag** shader provides a slider to control how much bump should be emphasized for display. The parameter is exposed as a UI slider as follows:

```
//! input | BumpWeight | Bump Weight | float    | slider   |
1.0,0.0,10.0
```

The last three values are the default value, minimum value and maximum value for the slider.

After the section of instructions embedded in the comments, you simply define the function previously declared for Mari. This function is written in plain GLSL, with the exception of some functions that are defined in **Surface.glslh**.

For example, you could access **height map** or **texture** information as follows:

```
float h[4] = get_heights(texture, State.UV);
vec4 color  = texture_lookup(texture, State.UV);
```

Loading Shader Modules with Python

As previously mentioned, it's not good practice to place additional shader modules in the **Media/Shaders/Modules** folder. Mari loads all modules from this location, but it's better to keep custom shader modules outside of Mari's bundle directory.

The following lines of Python (adapted from the Python API example code in **Media/Scripts/examples/register_shader.py**) registers the custom shader module in Mari:

```
# Register the code as a new custom shader module
try:
    mari.gl_render.registerCustomShaderModule(module_code)
    mari.utils.message('Registered my XYZ module. Open a project,
then Add shader module->XYZ') # Where XYZ is the UI name you gave
your shader.
except RuntimeError:
    mari.utils.message('Please close any open projects and try
again')
```

Fixing GLSL Compilation Errors

Mari submits the GLSL code for compilation and if the code fails to compile, Mari crashes. To debug your code, you need to consult the Mari log file, where the compilation error is logged. For example, if we introduce the following syntax error just after the start of the function:

```
    vvvec4 blend  = texture_lookup(texture, State.UV);
```

MariLog.txt contains the following compilation error information:

```
Debug : [    MriModularShaderObject.cpp: 283] : ===Compile Error
(file:///workspace/fanner/mari1.2dev/Apps/Mari/objects/linux-64-
x86-release-410/Bundle/Media/Shaders/Modules/Sub_Diffuse.frag)===
Debug : [          MriModularShader.cpp: 165] : ***COMPILER ERROR
(Vertex ModularShader):
Debug : [          MriModularShader.cpp: 166] : 0(70) : warning
C7551: OpenGL first class arrays require #version 120
0(96) : error C0000: syntax error, unexpected '=' at token "="
0(96) : error C0501: type name expected at token "="
0(96) : warning C7022: unrecognized profile specifier "blend"
0(96) : warning C7022: unrecognized profile specifier "vvvec4"
0(96) : error C0000: syntax error, unexpected ',' at token ","
0(96) : error C0501: type name expected at token ","
0(96) : warning C7022: unrecognized profile specifier "texture"
0(96) : warning C7022: unrecognized profile specifier "State"

Warning : error: can't compile shader objects
```

You can see that the syntax error is introduced on line 13, but the compiler

log indicates that the unrecognized specifier **vvvec4** is on line 96. The line difference is a result of Mari inserting the shader code at the appropriate place in a larger shader file. When searching for compilation bugs, be aware that the line numbers listed differ from your shader module lines. In most cases the shader that Mari compiles is included in the log with its own line numbers.

Mipmap Behavior

For **Environment Cube** shader modules, **.dds** files with missing mipmaps or partial mipchains continue to load, but the behavior for these chains is slightly different. Mari disregards all the other mipmaps in the chain; instead, it regenerates the chain from the top level mip.

In this way, Mari continues to load partial mipmap chains in a workable manner for **.dds** files.

Learning More

In addition to the above example, it's recommended that you:

- Study the Python API examples and documentation for working with shaders in **Media/Scripts/examples/register_shader.py** and **Media/Scripts/shader_baking.py**.
- Read the documentation for the shader-related API modules **mari.GLRender** and **mari.Shader**.
- Examine the HSV example shader in **Media/Scripts/examples/hsv_shader.frag**.
- Explore the other shader modules located in **Media/Shaders/modules** for more examples.