RV 3.12: Reference Manual

Tweak Software http://www.tweaksoftware.com

September 28, 2011

Contents

1 Overview **3**
1.1 The Big Picture . . . 3
1.2 Drawing . . . 4
1.3 Menus . . . 4
1.4 Interfacing with the Environment . . . 4
1.5 Setting and Getting the Image Processing Graph State . . . 5
1.5.1 Addressing Properties . . . 6
1.5.2 User Defined Properties . . . 6
1.6 Getting Information From Images . . . 9

2 Python **10**
2.1 Calling Mu From Python . . . 10
2.2 Calling Python From Mu . . . 11

3 Event Handling **12**
3.1 Binding an Event . . . 14
3.2 Keyboard Events . . . 14
3.3 Pointer (Mouse) Events . . . 15
3.4 The Render Event . . . 16
3.5 Remote Networking Events . . . 17
3.6 Internal Events . . . 17
3.6.1 File Changed Event . . . 17
3.6.2 Incoming Source Path Event . . . 17
3.6.3 Missing Images . . . 19

4 Using Qt in Mu **20**
4.1 Signals and Slots . . . 21
4.2 Inheriting from Qt Classes . . . 22

5 Modes and Widgets **24**
5.1 Outline of a Mode . . . 25
5.2 Outline of a Widget . . . 26

6 Package System **27**
6.1 `rvpkg` Command Line Tool . . . 27
6.1.1 Getting a List of Available Packages . . . 27
6.1.2 Getting Information About the Environment . . . 28
6.1.3 Getting Information About a Package . . . 28
6.1.4 Adding a Package to a Support Area . . . 28
6.1.5 Removing a Package from a Support Area . . . 28
6.1.6 Installing and Uninstalling Available Packages . . . 29
6.1.7 Combining Add and Install for Automated Installation . . . 29
6.1.8 Overrideing Default Optional Package Load Behavior . . . 29

6.2 Package File Contents 29
6.3 `PACKAGE` Format 29
6.4 Package Management Configuration Files 32
6.5 Developing a New Package 32
6.5.1 Older Package Files (.zip) 33
6.5.2 Using the Mode Manager While Developing 33
6.5.3 Using `-debug mu` 33
6.5.4 The Mu API Documentation Browser 33
6.6 Loading Versus Installing and User Override 34
6.6.1 Optional Packages 34

7 A Simple Package **35**
7.1 How Menus Work 37
7.2 A Menu in MyStuffMode 38
7.3 Finishing up 40

8 The Custom Matte Package **41**
8.1 Creating the Package 41
8.2 The Custom Matte File 41
8.3 Parsing the Matte File 41
8.4 The CustomMattesMode Constructor 44
8.5 Rendering the Matte 47
8.6 The Finished custom_mattes.mu File 49

9 Automated Color and Viewing Management **52**
9.1 The `new-source` Event 53
9.2 The default `new-source` behavior 53
9.3 Breakdown of `sourceSetup()` in the `source_setup` Package 54
9.4 Setting up 3D and Channel LUTs 55
9.5 Building a Package For Color Management 55

10 Network Communication **57**
10.1 Example Code 57
10.1.1 Using rvshell 57
10.1.2 Using rvNetwork.py 58
10.2 TwkQtChat Library 58
10.3 The Protocol 59
10.3.1 Standard Messages 59
10.3.2 Data Messages 62

11 Webkit JavaScript Integration **64**
11.1 Executing Mu or Python from JavaScript 64
11.2 Getting Event Call Backs in JavaScript 64
11.3 Using the `webview` Example Package 66

12 Hierarchical Preferences **68**

13 Node Reference **69**

Chapter 1

Overview

RV comes with the source code to its user interface. The code is written in a language called Mu which is not difficult to learn if you know Python, MEL, or most other computer languages used for computer graphics. As of 3.12, RV can also use Python in a nearly interchangable manner.

If you are completely unfamiliar with programming, you may still glean information about how to customize RV in this manual; but the more complex tasks like creating a special overlay or slate for RVIO or adding a new heads-up widget to RV might be difficult to understand without help from someone more experienced.

This manual does not assume you know Mu to start with, so you can dive right in. For Python, some assumptions are made. The chapters are organized with specific tasks in mind.

The reference chapters contain detailed information about various internals that you can modify from the UI.

Using the RV file format (.rv) is detailed in the User Manual.

1.1 The Big Picture

RV is two different pieces of software: the core (written in C++) and the interface (written in Mu and Python). The core handles the following things:

- Image, Movie, and Audio I/O
- Caching Images and Audio
- Tracking Dependencies Among Image and Audio Operations
- Basic Image Processing in Software
- Rendering Images
- Feeding Audio to Audio Output Devices

The interface — which is available to be modified — is concerned with the following:

- Handling User Events
- Rendering Additional Information and Heads-up Widgets
- Setting and Getting State in the Image Processing Graph
- Interfacing to the Environ
- ment
- Handling User Defined Setup of Incoming Movies/Images/Audio

- High Level Features

RVIO shares almost everything with RV including the UI code (if you want it to). However it will not launch a GUI so its UI is normally non-existent. RVIO does have additional hooks for modification at the user level: overlays and leaders. Overlays are Mu scripts which allow you to render additional visual information on top of rendered images before RVIO writes them out. Leaders are scripts which generate frames from scratch (there is nothing rendered under them) and are mainly there to generate customized flexible slates automatically.

1.2 Drawing

In RV's user interface code or RVIO's leader and overlays its possible draw on top of rendered frames. This is done using the industry standard API OpenGL. There are Mu modules which implement OpenGL 1.1 functions including the GLU library. In addition, there is a module which makes it easy to render true type fonts as textures (so you can scale, rotate, and composite characters as images). For Python there is PyOpenGL and related modules.

Mu has a number of OpenGL friendly data types include native support for 2D and 3D vectors and dependently typed matrices (e.g., float[4,4], float[3,3], float[4,3], etc). The Mu GL modules take the native types as input and return them from functions, but you can use normal GL documentation and man pages when programming Mu GL. In this manual, we assume you are already familiar with OpenGL. There are many resources available to learn it in a number of different programming languages. Any of those will suffice to understand it.

1.3 Menus

The menu bar in an RV session window is completely controlled (and created) by the UI. There are a number of ways you can add menus or override and replace the existing menu structure.

Adding one or more custom menus to RV is a common customization. This manual contains examples of varying complexity to show how to do this. It is possible to create static menus (pre-defined with a known set of menu items) or dynamic menus (menus that are populated when RV is initialized based on external information, like environment variables).

1.4 Interfacing with the Environment

One of the most important customizations you can make to RV is making it aware of the environment its running in.

For example, it is common practice in big post-production companies (primarily using Linux) to provide environment variables that indicate a show, shot and sequence that an artist is currently working on. A user might have two shell windows open: one in the environment of shot "A" and one in shot "B". Starting RV in the shot "A" environment might cause it to show different menus or automatically select a particular display LUT. These types of facility-context-dependent customizations can save artists a lot of time and in some cases relieve them from having to make decisions about color or finding the default location of certain files.

Following sections of this manual show a number of ways that you can customize RV for facility-wide use as well as personal use. We highly recommend facility-wide customizing because some institutional decisions can be hard coded there — especially those concerning how color should be managed.

Note that the use of Python in RV is nearly identical to Mu so other than the libraries and syntax, the mechanisms used to control RV are the same in the two languages.

Here's an example in Mu of how to get environment variables from the operating system with default values when the variable is not defined. This will work on all platforms:

```
\: show_environment (void;)
{
    use system;  // this is where getenv() lives
    let shot = getenv("SHOT", "noshot"),  // defaults to noshot
        show = getenv("SHOW", "noshow");  // defaults to noshow

    print("SHOT = %s, SHOW = %s\n" % (shot, show));
}
```

Listing 1.1: Example Use of Environment Variables

Its also possible to open a process stream to an external program and read its output. This example calls the unix ls command on a file (or directory) that is passed into the function and returns the output as a string. On windows you would need to use an appropriate command path and arguments for that platform. This example is fairly advanced:

```
\: get_ls_output (string; string file_to_ls)
{
    use io; // this is where the streams live

    let cmd = process("/bin/ls", ["-al", file_to_ls], int64.max);
    istream lsout = cmd.out();
    cmd.close_in();

    osstream result; // osstream is an output string stream

    while (true)
    {
        let line = read_line(lsout, '\n');
        if (line == "") break;
        print(result, "%s\n" % line);
    }

    cmd.close();
    return string(result);
}
```

Listing 1.2: Example Use of Process Stream

1.5 Setting and Getting the Image Processing Graph State

The UI needs to communicate with the core part of RV. This is done in two ways: by calling special command functions (commands) which act directly on the core (e.g. `play()` causes it to start playing), or by setting variables in the underlying image processing graph which control how images will be rendered.

Inside each session there is a *directed acyclic graph* (DAG) which determines how images and audio will be evaluated for display. The DAG is composed of *nodes* which are themselves collections of *properties*.

A node is something that produces images and/or audio as output from images and audio inputs (or no inputs in some cases). An example from RV is the *color* node; the color node takes images as input and produces images that are copies of the input images with the hue, saturation, exposure, and contrast potentially changed.

A *property* is a state variable. The node's properties as a whole determine how the node will change its inputs to produce its outputs. You can think of a node's properties as *parameters* that change its behavior.

RV's session file format (.rv file) stores all of the nodes associated with a session including each node's properties. So the DAG contains the complete state of an RV session. When you load an .rv file into RV, you create a new DAG based on the contents of the file. Therefore, *to change anything in RV that affects how an image looks, you must change a property in some node in its DAG.*

There are a few commands which RV provides to get and set properties: `setIntProperty()`, `getIntProperty()`, `setFloatProperty()`, `getFloatProperty()`, `setStringProperty()`, and `getStringProperty()`. These are available in both Mu and Python.

Finally, there is one last thing to know about properties: they are arrays of values. So a property may contain zero values (its empty) or one value or an array of values. The get and set functions above all deal with arrays of numbers even when a property only has a single value.

1.5.1 Addressing Properties

A full property name has three parts: the node name, the component name, and the property name. These are concatenated together with dots like *nodename.componentname.propertyname*. Each property has its own type which can be set and retrieved with one of the set or get functions. You must use the correct get or set function to access the property. For example, to set the display gamma, which is part of the "display" node, you need to use setFloatProperty() like so in Mu:

```
setFloatProperty("display.color.gamma", float[] {2.2, 2.2, 2.2}, true)
```

or in Python:

```
setFloatProperty("display.color.gamma",  [2.2, 2.2, 2.2], True)
```

In this case the value is being set to 2.2.

In an RV session, some node names will vary per the source(s) being displayed and some will not. Figure 1.1 shows a pipeline diagram for one possible configuration and indicates which are per-source (duplicated) and which are not.

At any point in time, a subset of the graph is active. For example if you have three sources in a session and RV is in sequence mode, at any given frame only one source branch will be active. There is a second way to address nodes in RV: by their types. This is done by putting a hash (#) in front of the type name. Addressing by node type will affect all of the currently active nodes of the given type. For example, a property in the color node is exposure which can be addressed directly like this in Mu:

```
color.color.exposure
```

or using the type name like this:

```
#RVColor.color.exposure
```

When the type name syntax is used, and you use one of the set or get functions on the property, only nodes that are currently active will be considered. So in this case, if we were to set the exposure using type-addressing:

```
setFloatProperty("#RVColor.color.exposure", float[] {2.0, 2.0, 2.0}, true)
```

or in Pythion:

```
setFloatProperty("#RVColor.color.exposure", [2.0, 2.0, 2.0], True)
```

It would affect any node of type `RVColor` that is currently active. In sequence mode (i.e. the default case), only one RVColor node is usually active at a time (the one belonging to the source being viewed at the current frame). In stack mode, the RVColor nodes for all of the sources could be active. In that case, they will all have their exposure set. In the UI, properties are almost exclusively addressed in this manner so that making changes affects the currently visible sources only. See figure 1.2 for a diagrammatic explanation.

Chapter 13 has all the details about each node type.

1.5.2 User Defined Properties

Its possible to add your own properties when creating an RV file from scratch or from the user interface code. The `newProperty()` function docs.

Why would you want to do this? There are a few reasons to add a user defined property:

1. You wish to save something in a session file that was created interactively by the user.

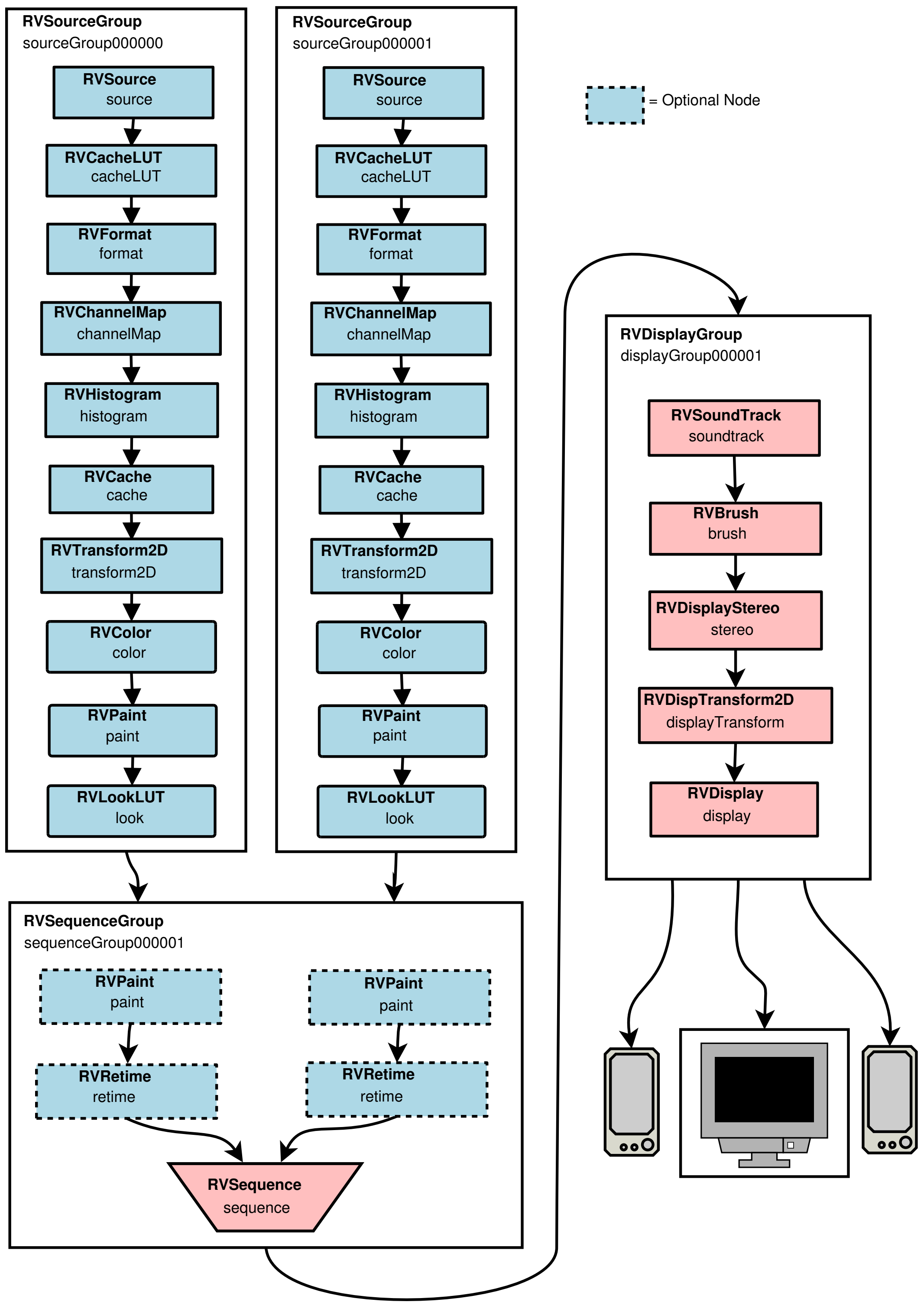

Figure 1.1: Conceptual diagram of RV Image and Audio Processing Graph for a session with a single sequence. Nodes are shown with both the type name in bold and the instance name underneath. Blue nodes are duplicated per-source in this session configuration and the instance names are numbered.

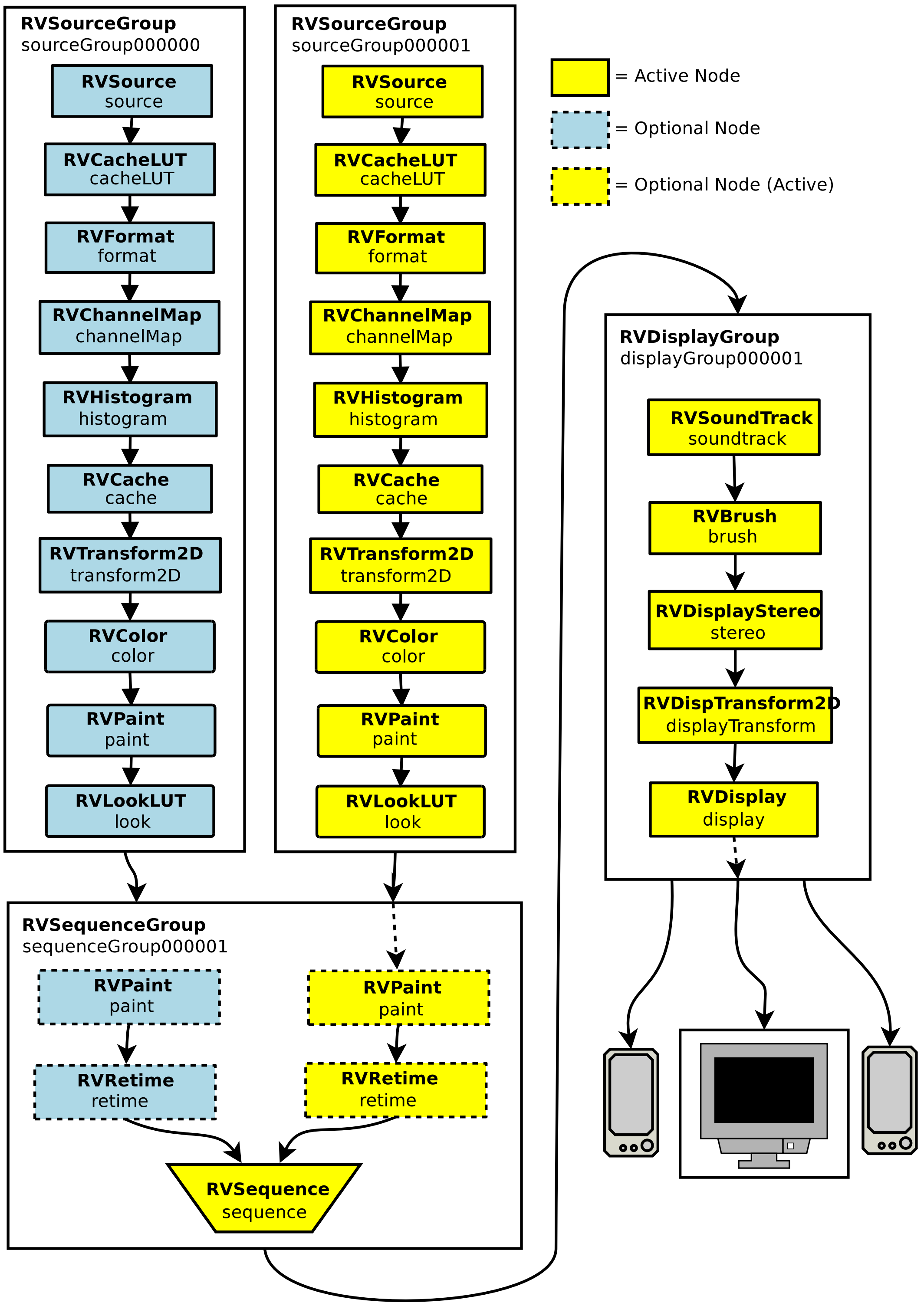

Figure 1.2: Active Nodes in the Image Processing Graph. The active nodes are those nodes which contribute to the rendered view at any given frame. In this configuration, when the sequence is active, there is only one source branch active at any given frame (the yellow nodes). By addressing nodes using their type name, you can affect only active nodes with that type.

Command	Description
`sourceAtPixel`	Given a point in the view, returns a structure with information about the source(s) underneath the point.
`sourcesRendered`	Returns information about all sources rendered in the current view (even those that may have been culled).
`sourceLayers`	Given the name of a source, returns the layers in the source.
`sourceGeometry`	Given the name of a source, returns the geometry (bounding box) of that source.
`sourceMedia`	Given the name of a source, returns the a list of its media files.
`sourcePixelValue`	Given the name of a source and a coordinate in the image, returns an RGBA pixel value at that coordinate. This function may convert chroma image pixels to Rec709 primary RGB in the process.
`sourceAttributes`	Given the name of a source and optionally the name a particular media file in the source, returns an array of tuples which contain attribute names and values.
`sourceStructure`	Given the name of a source and optionally the name a particular media file in the source, returns information about image size, bit depth, number of channels, underlying data type, and number of planes in the image.
`sourceDisplayChannelNames`	Given the name of a source, returns an array of channel names current being displayed.

Table 1.1: Command Functions for Querying Displayed Images

2. You're generating session files from outside RV and you want to include additional information (e.g. production tracking, annotations) which you'd like to have available when RV plays the session file.

Some of the packages that come with RV show how to implement functionality for the above.

1.6 Getting Information From Images

RV's UI often needs to take actions that depend on the context. Usually the context is the current image being displayed. Table 1.1 shows the most useful command functions for getting information about displayed images.

For example, when automating color management, the color space of the image or the origin of the image may be required to determine the best way to view the image (e.g., for a certain kind of DPX file you might want to use a particular display or file LUT). The color space is often stored as image attribute. In some cases, image attributes are misleading–for example, a well known 3D software package renders images with incorrect information about pixel aspect ratio—usually other information in the image attributes coupled with the file name and origin are enough to make a good guess.

Chapter 2

Python

As of RV 3.12 you can use Python in RV in conjunction with Mu or in place of it. Its even possible to call Python commands from Mu and vice versa. So in answer to the question: which language should I use to customize RV? The answer is whichever you like. At this point we recommend using Python.

There are some slight differences that need to be noted when translating code between the two languages:

In Python the modules names required by RV are the same as in Mu. As of this writing, these are commands, rvtypes, and rvui. However, the Python modules all live in the rv package. So while in Mu you can:

```
use commands
```

or

```
require commands
```

to make the commands visible in the current namespace. In Python you need to include the package name:

```
from rv.commands import *
```

or

```
import rv.commands
```

Pythonistas will know how all the permutations of the above.

2.1 Calling Mu From Python

Its possible to call Mu code from Python, but in practice you will probably not need to do this unless you need to interface with existing packages written in Mu.

To call a Mu function from Python, you need to import the MuSymbol type from the pymu module. In this example, the play function is imported and called F on the Python side. F is then exectued:

```
from pymu import MuSymbol
F = MuSymbol("commands.play")
F()
```

If the Mu function has arguments you supply them when calling. Return values are automatically converted between languages. The conversions are indicated in .

```
from pymu import MuSymbol
F = MuSymbol("commands.isPlaying")
G = MuSymbol("commands.setWindowTitle")
if F() == True:
    G("PLAYING")
```

Python Type	**Converts to Mu Type**	**Converts To Python Type**	
Str or Unicode	`string`	Unicode string	Normal byte strings and unicode strings are both converted to Mu's unicode string. Mu strings always convert to unicode Python strings.
Int	`int`, `short`, or `byte`	Int	
Long	`int64`	Long	
Float	`foat` or `half` or `double`	Float	Mu double values may lose precision. Python float values may lose precision if passed to a Mu function that takes a half.
Bool	`bool`	Bool	
(Float, Float)	`vector float[2]`	(Float, Float)	Vectors are represented as tuples in Python
(Float, Float, Float)	`vector float[3]`	(Float, Float, Float)	
(Float, Float, Float, Float)	`vector float[4]`	(Float, Float, Float, Float)	
Tuple	tuple	Tuple	Tuple elements each convert independently
List	`type[]` or `type[N]`	List	Arrays (Lists) convert back and forth
Dictionary	Class	Dictionary	Class labels become dictionary keys

Table 2.1: Mu-Python Value Conversion

Once a MuSymbol object has been created, the overhead to call it is minimal. All of the Mu commands module is imported on start up or reimplemented as native CPython in the Python `rv.commands` module so you will not need to create MuSymbol objects yourself; just import `rv.commands` and use the pre-existing ones.

When a Mu function parameter takes a class instance, a Python dictionary can be passed in. When a Mu function returns a class, a dictionary will be returned. Python dictionaries should have string keys which have the same names as the Mu class fields and corresponding values of the correct types.

For example, the Mu class `Foo { int a; float b; }` as instantiated as `Foo(1, 2.0)` will be converted to the Python dictionary `{'a' : 1, 'b' : 2.0}` and vice versa.

2.2 Calling Python From Mu

In order to call Python objects from Mu you need to use the MuPy module. This implements a small subset of the CPython API. You can see documentation for this module in the Mu Command API Browser under the Help menu.

Chapter 3

Event Handling

Aside from rendering, the most important function of the UI is to handle events. An event can be triggered by any of the following:

- The mouse pointer moved or a button on the mouse was pressed
- A key on the keyboard was pressed or released
- The window needs to be re-rendered
- A file being watched was changed
- The user became active or inactive
- A supported device (like the apple remote control) did something
- An internal event like a new source or session being created has occurred

Each specific event has a name may also have extra data associated with it in the form of an event object. To see the name of an event (at least for keyboard and mouse pointer events) you can select the Help→Describe... which will let you interactively see the event name as you hit keys or move the mouse. You can also use Help→Describe Key.. to see what a specific key is bound to by pressing it.

Table 3.1 shows the basic event type prefixes.

When an event is generated in RV, the application will look for a matching event name in its bindings. The bindings are tables of functions which are assigned to certain event names. The tables form a stack which can be pushed and popped. Once a matching binding is found, RV will execute the function.

When receiving an event, all of the relevant information is in the Event object. This object has a number of methods which return information depending on the kind of event.

Event Prefix	Description
`key-down`	Key is being pressed on the keyboard
`key-up`	Key is being released on the keyboard
`pointer`	The mouse moved, button was pressed, or the pointer entered (or left) the window
`dragdrop`	Something was dragged onto the window (file icon, etc)
`render`	The window needs updating
`user`	The user's state changed (active or inactive, etc)
`remote`	A network event

Table 3.1: Event Prefixes for Basic Device Events

Method	**Events**	**Description**
`pointer (Vec2;)`	pointer-* dragdrop-*	Returns the location of the pointer relative to the view.
`relativePointer (Vec2;)`	pointer-* dragdrop-*	Returns the location of the pointer relative to the current widget or view if there is none.
`reference (Vec2;)`	pointer-* dragdrop-*	Returns the location of initial button mouse down during dragging.
`domain (Vec2;)`	pointer-* render-* dragdrop-*	Returns the size of the view.
`subDomain (Vec2;)`	pointer-* render-* dragdrop-*	Returns the size of the current widget if there is one. relativePointer() is positioned in the subDomain().
`buttons (int;)`	pointer-* dragdrop-*	Returns an int or'd from the symbols: `Button1`, `Button2`, and `Button3`.
`modifiers (int;)`	pointer-* key-* dragdrop-*	Returns an int or'd from the symbols: `None, Shift, Control, Alt, Meta, Super, CapLock, NumLock, ScrollLock`.
`key (int;)`	key-*	Returns the "keysym" value for the key as an int
`name (string;)`	any	Returns the name of the event
`contents (string;)`	internal events dragdrop-*	Returns the string content of the event if it has any. This is normally the case with internal events like new-source, new-session, etc. Pointer, key, and other device events do not have a contents() and will throw if its called on them. Drag and drop events return the data associated with them. Some render events have contents() indicating the type of render occuring.
`sender (string;)`	any	Returns the name of the sender
`contentType (int;)`	dragdrop-*	Returns an int describing the contents() of a drag and drop event. One of: `UnknownObject, BadObject, FileObject, URLObject, TextObject`.
`timeStamp (float;)`	any	Returns a float value in seconds indicating when the event occured
`reject (void;)`	any	Calling this function will cause the event to be send to the next binding found in the event table stack. Not calling this function stops the propagation of the event.
`setReturnContents (void; string)`	internal events	Events which have a contents may also have return content. This is used by the remote network events which can have a response.

Table 3.2: Event Object Methods. Python methods have the same names and return the same value types.

3.1 Binding an Event

In Mu (or Python) you can bind an event using any of the `bind()` functions. The most basic version of `bind()` takes the name of the event and a function to call when the event occurs as arguments. The function argument (which is called when the event occurs) should take an `Event` object as an argument and return nothing (`void`). Here's a function that prints hello in the console every time the "j" key is pressed:[1]

```
\: my_event_function (void; Event event)
{
    print("Hello!\n");
}

bind("key-down--j", my_event_function);
```

or in Python:

```
def my_event_function (event):
    print "Hello!"

bind("default", "global", "key-down--j", my_event_function);
```

There are more complicated bind() functions to address binding functions in specific event tables (the Python example above is using the most general of these). Currently RV's user interface has one default global event table an couple of other tables which implement the parameter edit mode and help modes.

Many events provide additional information in the event object. Our example above doesn't even use the event object, but we can change it to print out the key that was pressed by changing the function like so:

```
\: my_event_function (void; Event event)
{
    let c = char(event.key());
    print("Key pressed = %c\n" % c);
}
```

or in Python:

```
def my_event_function (event):
    c = event.key()
    print "Key pressed = %s\n" % c
```

In this case, the `Event` object's `key()` function is being called to retrieve the key pressed. To use the return value as a key it must be cast to a `char`. In Mu, the `char` type holds a single unicode character. In Python a string is used.

See the section on the Event class to find out how to retrieve information from it. At this point we have not talked about *where* you would bind an event; that will be addressed in the customization sections.

3.2 Keyboard Events

There are two keyboard events: `key-down` and `key-up`. Normally the `key-down` events are bound to functions. The `key-up` events are necessary only in special cases.

The specific form for key down events is `key-down--`*`something`* where *something* uniquely identifies both the key pressed and any modifiers that were active at the time.

So if the "a" key was pressed the event would be called: `key-down--a`. If the control key were held down while hitting the "a" key the event would be called `key-down--control--a`.

[1]If this is the first time you've seen this syntax, its defining a Mu function. The first two characters `\:` indicate a function definition follows. The name comes next. The arguments and return type are contained in the parenthesis. The first identifier is the return type followed by a semicolon, followed by an argument list.
E.g, `\: add (int; int a, int b) { return a + b; }`

There are five modifiers that may appear in the event name: `alt`, `caplock`, `control`, `meta`, `numlock`, `scrolllock`, and `shift` in that order. The shift modifier is a bit different than the others. If a key is pressed with the shift modifier down and it would result in a different character being generated, then the shift modifier will not appear in the event and instead the result key will. This may sound complicated but these examples should explain it:

For `control + shift + A` the event name would be `key-down--control--A`. For the "`*`" key (shift + 8 on American keyboards) the event would be `key-down--*`. Notice that the shift modifier does not appear in any of these. However, if you hold down shift and hit enter on most keyboards you will get `key-down--shift--enter` since there is no character associated with that key sequence.

Some keys may have a special name (like enter above). These will typically be spelled out. For example pressing the "home" key on most keyboards will result in the event `key-down--home`. The only way to make sure you have the correct event name for keys is to start RV and use the Help→Describe... facility to see the true name. Sometimes keyboards will label a key and produce an unexpected event. There will be some keyboards which will not produce an event all for some keys or will produce a unicode character sequence (which you can see via the help mechanism).

3.3 Pointer (Mouse) Events

The mouse (called pointer from here on) can produce events when it is moved, one of its buttons is pressed, an attached scroll wheel is rotated, or the pointer enters or leaves the window.

The basic pointer events are `move`, `enter`, `leave`, `wheelup`, `wheeldown`, `push`, `drag`, and `release`. All but `enter` and `leave` will also indicate any keyboard modifiers that are being pressed along with any buttons on the mouse that are being held down. The buttons are numbered 1 through 5. For example if you hold down the left mouse button and movie the mouse the events generated are:

```
pointer-1--push
pointer-1--drag
pointer-1--drag
...
pointer-1-release
```

Pointer events involving buttons and modifiers always come in there parts: push, drag and release. So for example if you press the left mouse, move the mouse, press the shift key, move the mouse, release everything you get:

```
pointer-1--push
pointer-1--drag
pointer-1--drag
...
pointer-1-release
pointer-1--shift--push
pointer-1--shift--drag
pointer-1--shift--drag
...
pointer-1--shift--release
```

Notice how the first group without the shift is released before starting the second group with the shift even though you never released the mouse button. For any combination of buttons and modifiers, there will be a push-drag-release sequence that is cleanly terminated.

It is also possible to hold multiple mouse buttons and modifiers down at the same time. When multiple buttons are held (for example, button 1 and 2) they are simply both included (like the modifiers) so for buttons 1 and 2 the name would be `pointer-1-2--push` to start the sequence.

The mouse wheel behaves more like a button: when the wheel moves you get only a `wheelup` or `wheeldown` event indicating which direction the wheel was rotated. The buttons and modifiers will be applied to the event name if they are held down. Usually the motion of the wheel on a mouse will not be smooth and the event will be emitted whenever the wheel "clicks". However, this is completely a function of the hardware so you may need to experiment with any particular mouse.

There are three more pointer events that can be generated. When the mouse moves with no modifiers or buttons held down it will generate the event `pointer--move`. When the pointer enters the view `pointer--enter` is generated and when it leaves `pointer--leave`. Something to keep in mind: when the pointer leaves the view and the device is no longer in focus on the RV window, any modifiers or buttons the user presses will not be known to RV and will not generate events. When the pointer returns to the view it may have modifiers that became active when out-of-focus. Since RV cannot know about these modifiers and track them in a consistent manner (at least on X Windows) RV will assume they do not exist.

Pointer events have additional information associated with them like the coordinates of the pointer or where a push was made. These will be discussed later.

3.4 The Render Event

The UI will get a render event whenever it needs to be updated. When handling the render event, a GL context is set up and you can call any GL function to draw to the screen. The event supplies additional information about the view so you can set up a projection.

At the time the render event occurs, RV has already rendered whatever images need to be displayed. The UI is then called in order to add additional visual objects like an on-screen widget or annotation.

Here's a render function that draws a red polygon in the middle of the view right on top of your image.

```
\: my_render (void; Event event)
{
    let domain = event.domain(),
        w      = domain.x,
        h      = domain.y,
        margin = 100;

    use gl;
    use glu;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, w, 0, h);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Big red polygon
    glColor(Color(1,0,0,1));
    glBegin(GL_POLYGON);
    glVertex(margin, margin);
    glVertex(w-margin, margin);
    glVertex(w-margin, h-margin);
    glVertex(margin, h-margin);
    glEnd();
}
```

Listing 3.1: Example Render Function

Note that for Python, you will need to use the PyOpenGL module or bind the symbols in the gl Mu module manually in order to draw in the render event.

The UI code already has a function called `render()` bound the render event; so this function basically disables existing UI rendering.

3.5 Remote Networking Events

RV's networking generates a number of events indicating the status of the network. In addition, once a connection has been established, the UI may generate sent to remote programs, or remote programs may send events to RV. These are typically uniquely named events which are specific to the application that is generating and receiving them.

For example the sync mechanism generates a number of events which are all named `remote-sync-`*`something`*.

3.6 Internal Events

Some events will originate from RV itself. These include things like `new-source` or `new-session` which include information about what changed. The most useful of these is `new-source` which can be used to manage color and other image settings between the time a file is loaded and the time it is first displayed. (See Color Management Section). Other internal events are functional, but are placeholders which will become useful with future features.

The current internal events are listed in table 3.3.

3.6.1 File Changed Event

It is possible to watch a file from the UI. If the watched file changes in any way (modified, deleted, moved, etc) a file-changed event will be generated. The event object will contain the name of the watched file that changed. A function bound to file-changed might look something like this:

```
\: my_file_changed (void; Event event)
{
    let file = event.contents();
    print("%s changed on disk\n" % file);
}
```

In order to have a file-changed event generated, you must first have called the command function `watchFile()`.

3.6.2 Incoming Source Path Event

This event is sent when the user has selected a file or sequence to load from the UI or command line. The event contains the name of the file or sequence. A function bound to this event can change the file or sequence that RV actually loads by setting the return contents of the event. For example, you can cause RV to check and see if a single file is part of a larger sequence and if so load the whole sequence like so:

```
\: load_whole_sequence (void; Event event)
{
    let file        = event.contents(),
        (seq,frame) = sequenceOfFile(event.contents());

    if (seq != "") event.setReturnContent(seq);
}

bind("incoming-source-path", load_whole_sequence);
```

or in Python:

```
def load_whole_sequence (event):
{
    file = event.contents();
    (seq,frame) = rv.commands.sequenceOfFile(event.contents());

```

Event	Event.(data/contents)	Description
`render`		Main view render
`layout`		Main view layout used to handle view margin changes
`new-session`		A new session was created or loaded into RV
`new-source`	*nodename*`;;RVSource;;`*filename*	A new source node was added
`source-modified`	*nodename*`;;RVSource;;`*filename*	An existing source was changed
`media-relocated`	*nodename*`;;`*oldmedia*`;;`*newmedia*	A movie, image sequence, audio file was swapped out
`source-media-set`	*nodename*`;;`*tag*	
`before-session-read`	*filename*	Session file is about to be read
`after-session-read`	*filename*	Session file was read
`before-session-write`	*filename*	Session file is about to be written
`after-session-write`	*filename*	Session file was just written
`before-session-write-copy`	*filename*	A copy of the session is about to be written
`after-session-write-copy`	*filename*	A copy of a session was just written
`before-session-deletion`		The session is about to be deleted
`before-graph-view-change`	*nodename*	The current view node is about to change.
`after-graph-view-change`	*nodename*	The current view node changed.
`new-node`	*nodename*	A new view node was created.
`before-progressive-loading`		Loading will start
`after-progressive-loading`		Loading is complete
`graph-layer-change`		**DEPRECATED** use after-graph-view-change
`frame-changed`		The current frame changed
`fps-changed`		Playback FPS changed
`play-start`		Playback started
`play-stop`		Playback stopped
`incoming-source-path`	*infilename*`;;`*tag*	A file was selected by the user for loading.
`missing-image`		An image could not be loaded for rendering
`cache-mode-changed`	`buffer` or `region` or `off`	Caching mode changed
`view-size-changed`		The viewing area size changed
`new-in-point`	*frame*	The in point changed
`new-out-point`	*frame*	The out point changed
`before-source-delete`	*nodename*	Source node will be deleted
`after-source-delete`	*nodename*	Source node was deleted
`before-node-delete`	*nodename*	View node will be deleted
`after-node-delete`	*nodename*	View node was deleted
`after-clear-session`		The session was just cleared
`after-preferences-write`		Preferences file was written by the Preferences GUI
`state-initialized`		Mu/Python init files read
`realtime-play-mode`		Playback mode changed to realtime
`play-all-frames-mode`		Playback mode changed to play-all-frames
`before-play-start`		Play mode will start
`mark-frame`	*frame*	Frame was marked
`unmark-frame`	*frame*	Frame was unmarked
`pixel-block`	Event.data()	A block of pixels was received from a remote connection
`graph-state-change`		A property in the image processing graph changed
`graph-node-inputs-changed`	*nodename*	Inputs of a top-level node added/removed/re-ordered
`range-changed`		The time range changed
`narrowed-range-changed`		The narrowed time range changed
`margins-changed`	*left right top bottom*	View margins changed
`view-resized`	*old-w new-w* `|` *old-h new-h*	Main view changed size
`preferences-show`		Pref dialog will be shown
`preferences-hide`		Pref dialog was hidden
`remote-eval`	*code*	Request to evaluate external Mu code
`py-remote-eval`	*code*	Request to evaluate external Python code
`remote-network-start`		Remote networking started
`remote-network-stop`		Remote networking stopped
`remote-connection-start`	*contact-name*	A new remote connection has been made
`remote-connection-stop`	*contact-name*	A remote connection has died
`remote-contact-error`	*contact-name*	A remote connection error occured while being established

Table 3.3: Internal Events

```
    if seq != "":
         event.setReturnContent(seq);
}

bind("default", "global", "incoming-source-path", load_whole_sequence, "Doc string");
```

3.6.3 Missing Images

Sometimes an image is not available on disk when RV tries to read. This is often the case when looking at an image sequence while a render or composite is ongoing. By default, RV will find a nearby frame to represent the missing frame if possible. The missing-image event will be sent once for each image which was expected but not found. The function bound to this event can render information on on the screen indicating that the original image was missing. The default binding display a message in the feedback area.

The missing-image event contains the domain in which rendering can occur (the window width and height) as well as a string of the form "*frame;source*" which can be obtained by calling the `contents()` function on the event object.

The default binding looks like this:

```
\: missingImage (void; Event event)
{
    let contents = event.contents(),
        parts = contents.split(";"),
        media = io.path.basename(sourceMedia(parts[1])._0);

    displayFeedback("MISSING: frame %s of %s"
                    % (parts[0], media), 1, drawXGlyph);
}

bind("missing-image", missingImage);
```

Chapter 4

Using Qt in Mu

Since version 3.8 RV has had limited Qt bindings in Mu. In 3.10 the number of available Qt classes has been greatly expanded. You can browse the Qt and other Mu modules with the documentation browser.

Using Qt in Mu is similar to using it in C++. Each Qt class is presented as a Mu class which you can either use directly or inherit from if need be. However, there are some major differences that need to be observed:

- *Not all Qt classes are wrapped in Mu.* Its a good idea to look in the documentation browser to see if a class is available yet.
- *Property names in C++ do not always match those in Mu.* Mu collects Qt properties at runtime in order to provide limited supported for unknown classes. So the set and get functions for the properties are generated at that time. Usually these names match the C++ names, but sometimes there are differences. In general, the Mu function to get a property called **foo** will be called `foo()`. The Mu function to set the foo property will be called `setFoo()`. [1]
- *Templated classes in Qt are not available in Mu.* Usually these are handled by dynamic array types or something analogous to the Qt class. In the case of template member functions (like `QWidget::findChild<>`) there may be an equivalent Mu version that operates slightly differently (like the Mu version `QWidget.findChild`).
- *The QString class is not wrapped (yet).* Instead, the native Mu string can be used anywhere a function takes a QString.
- *You cannot control widget destruction.* If you loose a reference to a QObject it will eventually be finalized (destroyed), but at an unknown time.
- *Some classes cannot be inherited from.* You can inherit from any QObject, QPainter, or QLayoutItem derived class except QWebFrame and QNetworkReply.
- *The signal slot mechanism is slightly different in Mu than C++.* It is currently not possible to make a new Qt signal, and slots do not need to be declared in a special way (but they do need to have the correct signatures to be connected). In addition, you are not required to create a QObject class to receive a signal in Mu. You can also connect a signal directly to a regular function if desired (as opposed to class member functions in C++).
- *Threading is not yet available.* The QThread class cannot be used in Mu yet.
- *Default parameter values are not yet translated into Mu.* This means that you must supply all arguments – even default arguments – when calling a Qt function in Mu.
- *Abstract Qt classes can be instantiated.* However, you can't really do anything with them.
- *Protected member functions are public.*

[1] A good example of this is the QWidget property **visible**. In C++ the get function is `isVisible()` whereas the Mu function is called `visible()`.

4.1 Signals and Slots

Possibly the biggest difference between the Mu and C++ Qt API is how signals and slots are handled. This discussion will assume knowledge of the C++ mechanism. See the Qt documentation if you don't know what signals and slots are.

Jumping right in, here is an example hello world MuQt program. This can be run from the mu-interp binary:

```
use qt;

\: clicked (void; bool checked)
{
    print("OK BYE\n");
    QCoreApplication.exit(0);
}

\: main ()
{
    let app    = QApplication(string[] {"hello.mu"}),
        window = QWidget(nil, Qt.Window),
        button = QPushButton("MuQt: HELLO WORLD!", window);

    connect(button, QPushButton.clicked, clicked);

    window.setSize(QSize(200, 50));
    window.show();
    window.raise();
    QApplication.exec();
}

main();
```

The main thing to notice in this example is the `connect()` function. A similar C++ version of this would look like this:

```
connect(button, SIGNAL(clicked(bool)), SLOT(myclickslot(bool)));
```

where `myclickslot` would be a slot function declared in a class. In Mu its not necessary to create a class to recieve a signal. In addition the `SIGNAL` and `SLOT` syntax is also unnecessary. However, it is necessary to exactly specify which signal is being referred to by passing its Mu function object directly. In this case `QPushButton.clicked`. The signal must be a function on the class of the first argument of `connect()`.

In Mu, any function which matches the signal's signature can be used to receive the signal. The downside of this is that some functions like sender() are not available in Mu. However this is easily overcome with partial application. In the above case, if we need to know who sent the signal in our clicked function, we can change its signature to accept the sender and partially apply it in the connect call like so:

```
\: clicked (void; bool checked, QPushButton sender)
{
    // do something with sender
}

\: main ()
{
    ...

    connect(button, QPushButton.clicked, clicked(,button));
}
```

And of course additional information can be passed into the clicked function by applying more arguments.

Its also possible to connect a signal to a class method in Mu if the method signature matches. Partial application can be used in that case as well. This is frequently the case when writing a mode which uses Qt interface.

4.2 Inheriting from Qt Classes

Its possible to inherit directly from the Qt classes in Mu and override methods. Virtual functions in the C++ version of Qt are translated as class methods in Mu. Non-virtual functions are regular functions in the scope of the class. In practice this means that the Mu Qt class usage is very similar to the C++ usage.

The following example shows how to create a new widget type that implements a drop target. Drag and drop is one aspect of Qt that requires inheritance (in C++ and Mu):

```
use qt;

class: MyWidget : QWidget
{
    method: MyWidget (MyWidget; QObject parent, int windowFlags)
    {
        // REQUIRED: call base constructor to build Qt native object
        QWidget.QWidget(this, parent, windowFlags);
        setAcceptDrops(true);
    }

    method: dragEnterEvent (void; QDragEnterEvent event)
    {
        print("drop enter\n");
        event.acceptProposedAction();
    }

    method: dropEvent (void; QDropEvent event)
    {
        print("drop\n");
        let mimeData = event.mimeData(),
            formats = mimeData.formats();

        print("--formats--\n");
        for_each (f; formats) print("%s\n" % f);

        if (mimeData.hasUrls())
        {
            print("--urls--\n");
            for_each (u; event.mimeData().urls())
                print("%s\n" % u.toString(QUrl.None));
        }

        if (mimeData.hasText())
        {
            print("--text--\n");
            print("%s\n" % mimeData.text());
        }

        event.acceptProposedAction();
        }
}
```

Things to note in this example: the names of the drag and drop methods matter. These are same names as used in C++. If you browser the documentation of a Qt class in Mu these will be the class methods. Only class methods can be overriden.

Chapter 5

Modes and Widgets

The user interface layer can augment the display and event handling in a number of different ways. For display, at the lowest level its possible to intercept the render event in which case you override all drawing. Similarily for event handling you can bind functions in the global event table possibly overwriting existing bindings and thus replace their functions.

At a higher level, both display and event handling can be done via Modes and Widgets. A Mode is a class which manages an event table independent of the global event table and a collection of functions which are bound in that table. In addition the mode can have a render function which is automatically called at the right time to augment existing rendering instead of replacing it. The UI has code which manages modes so that they may be loaded externally only when needed and automatically turned on and off.

Modes are further classified as being minor or major. The only difference between them is that a major mode will always get precedence over any minor mode when processing events and there can be only a single major mode active at a time. There can be many minor modes active at once. Most extensions are created by creating a minor mode. RV currently has a single basic major mode.

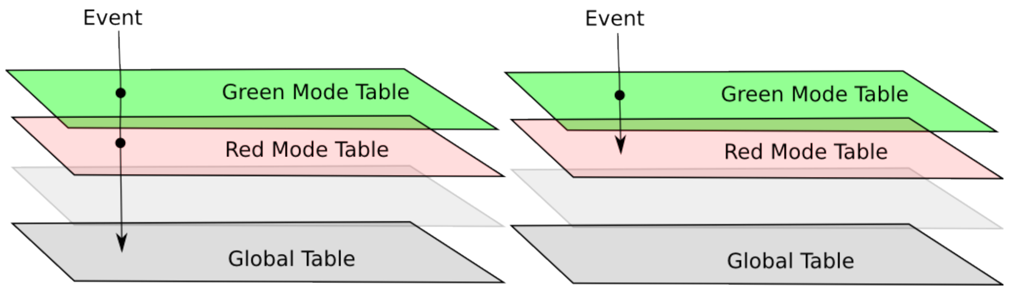

Figure 5.1: Event Propagation. Red and Green modes process the event. On the left the Red mode rejects the event allowing it to continue. On the right Red mode does not reject the event stopping the propagation.

By using a mode to implement a new feature or replace or augment an existing feature in RV you can keep your extensions seperate from the portion of the UI that ships with RV. In other words, you never need to touch the shipped code and your code will remain isolated.

A further refinement of a mode is a widget. Widgets are minor modes which operate in a constrainted region of the screen. When the pointer is in the region, the widget will receive events. When the pointer is outside the region it will not. Like a regular mode, a widget has a render function which can draw anywhere on the screen, but usually is constrainted to its input region. For example, the image info box is a widget as is the color inspector.

Multiple modes and widgets may be active at the same time.

5.1 Outline of a Mode

In order to create a new mode you need to create a module for it and derive your mode class from the `MinorMode` class in the `rvtypes` module. The basic outline which we'll put in a file called `new_mode.mu` looks like this:

```
use rvtypes;

module: new_mode {

class: NewMode : MinorMode
{
    method: NewMode (NewMode;)
    {
        init ("new-mode",
              [ global bindings ... ],
              [ local bindings ... ],
              Menu(...) );
    }
}

\: createMode (Mode;)
{
    return NewMode();
}

} // end of new_mode module
```

The function `createMode()` is used by the mode manager to create your mode without knowing anything about it. It should be declared in the scope of the module (not your class) and simply create your mode object and initialize it if that's necessary.

When creating a mode its necessary to call the `init()` function from within your constructor method. This function takes at least three arguments and as many as six. Chapter 7 goes into detail about the structure in more detail. Its declared like this in `rvtypes.mu`:

```
method: init (void;
              string name,
              BindingList globalBindings,
              BindingList overrideBindings,
              Menu menu = nil,
              string sortKey = nil,
              int ordering = 0)
```

The name of the mode is meant to be human readable. The `globalBindings` argument is a list of binding tuples which are applied on top of the existing global bindings. When the mode becomes inactive these bindings will go away. While the mode is active the underlying bindings are not accessible.

The `overrideBindings` argument is similar, but its bindings are layered over the global bindings and any other modes which are on. This means that the underlying global (or other mode) bindings can still be activated. In your event function you can reject an event which will cause rv to pass it on to bindings underneath yours. This technique allows you to augment an existing binding instead of replacing it.

The menu argument allows you to pass in a menu structure which is merged into the main menu bar. This makes it possible to add new menus and menu items to the existing menus.

Finally the sortKey and ordering arguments allow fine control over the order in which events are considered when multiple modes are active. Normally the name of the mode is used as the sorting key. By supplying a non-nil value you can use another string instead of the name. In the case that two modes have the same key the order is considered as well.

Again, see chapter 7 for more detailed information.

5.2 Outline of a Widget

A Widget looks just like a MinorMode declaration except you will derive from `Widget` instead of `MinorMode` and the base class `init()` function is simpler. In addition, you'll need to have a `render()` method (which is optional for regular modes).

```
use rvtypes;

module: new_widget {

class: NewWidget : Widget
{
    method: NewWidget (NewWidget;)
    {
        init ("new-widget",
              [ local bindings ... ] );
    }

    method: render (void; Event event)
    {
        ...
        updateBounds(min_point, max_point);
        ...
    }
}

\: createMode (Mode;)
{
    return NewWidget();
}

} // end of new_widget module
```

In the outline above, the function `updateBounds()` is called in the `render()` method. `updateBounds()` informs the UI about the bounding box of your widget. This function must be called by the widget at some point. If your widget can be interactively or procedurally moved, you will probably want to may want to call it in your `render()` function as shown (it does not hurt to call it often). The `min_point` and `max_point` arguments are `Vec2` types.

Chapter 6

Package System

With previous versions of RV we recommend directly hacking the UI code or setting up ad hoc locations in the `MU_MODULE_PATH` to place files.

For RV 3.6 or newer, we recommend using the new package system instead. The documentation in older versions of the reference manual is still valid, but we will no longer be using those examples. There are hardly any limitations to using the package system so no additional features are lost.

6.1 `rvpkg` Command Line Tool

The `rvpkg` command line tool makes it possible to manage packages from the shell. If you use rvpkg you do not need to use RV's preferences UI to install/uninstall add/remove packages from the file system. We recommend using this tool instead of manually editing files to prevent the necessity of keeping abreast of how all the state is stored in new versions.

The `rvpkg` tool can perform a superset of the functions available in RV's packages preference user interface.

Note: many of the below commands, including `install`, `uninstall`, and `remove` will look for the designated packages in the paths in the RV_SUPPORT_PATH environment variable. If the package you want to operate on is not in a path listed there, that path can be added on the command line with the `-include` option.

6.1.1 Getting a List of Available Packages

```
shell> rvpkg -list
```

-include *directory*	include directory as if part of `RV_SUPPORT_PATH`
-env	show `RV_SUPPORT_PATH` include app areas
-only *directory*	use directory as sole content of `RV_SUPPORT_PATH`
-add *directory*	add packages to specified support directory
-remove	remove packages (by name, rvpkg name, or full path to rvpkg)
-install	install packages (by name, rvpkg name, or full path to rvpkg)
-uninstall	uninstall packages (by name, rvpkg name, or full path to rvpkg)
-optin	make installed optional packages opt-in by default for all users
-list	list installed packages
-info	detailed info about packages (by name, rvpkg name, or full path to rvpkg)
-force	Assume answer is 'y' to any confirmations – don't be interactive

Table 6.1: `rvpkg` Options

Lists all packages that are available in the RV_SUPPORT_PATH directories. Typical output from rvpkg looks like this:

```
I L - 1.7 "Annotation" /SupportPath/Packages/annotate-1.7.rvpkg
I L - 1.1 "Documentation Browser" /SupportPath/Packages/doc_browser-1.1.rvpkg
I - O 1.1 "Export Cuts" /SupportPath/Packages/export_cuts-1.1.rvpkg
I - O 1.3 "Missing Frame Bling" /SupportPath/Packages/missing_frame_bling-1.3.rvpkg
I - O 1.4 "OS Dependent Path Conversion" /SupportPath/Packages/os_dependent_path_conversion_mode-1.4.rvpkg
I - O 1.1 "Nuke Integration" /SupportPath/Packages/rvnuke-1.1.rvpkg
I - O 1.2 "Sequence From File" /SupportPath/Packages/sequence_from_file-1.2.rvpkg
I L - 1.3 "Session Manager" /SupportPath/Packages/session_manager-1.3.rvpkg
I L - 2.2 "RV Color/Image Management" /SupportPath/Packages/source_setup-2.2.rvpkg
I L - 1.3 "Window Title" /SupportPath/Packages/window_title-1.3.rvpkg
```

The first three columns indicate installation status (I), load status (L), and whether or not the package is optional (O).

If you want to include a support path directory that is not in RV_SUPPORT_PATH, you can include it like this:

```
shell> rvpkg -list -include /path/to/other/support/area
```

To limit the list to a single support area:

```
shell> rvpkg -list -only /path/to/area
```

The -include and -only arguments may be applied to other options as well.

6.1.2 Getting Information About the Environment

You can see the entire support path list with the command:

```
shell> rvpkg -env
```

This will show alternate version package areas constructed from the RV_SUPPORT_PATH environment variable to which packages maybe added, removed, installed and uninstalled. The list may differ based on the platform.

6.1.3 Getting Information About a Package

```
shell> rvpkg -info /path/to/file.rvpkg
```

This will result in output like:

```
Name: Window Title
Version: 1.3
Installed: YES
Loadable: YES
Directory:
Author: Tweak Software
Organization: Tweak Software
Contact: an actual email address
URL: http://www.tweaksoftware.com
Requires:
RV-Version: 3.9.11
Hidden: YES
System: YES
Optional: NO
Writable: YES
Dir-Writable: YES
Modes: window_title
Files: window_title.mu
```

6.1.4 Adding a Package to a Support Area

```
shell> rvpkg -add /path/to/area /path/to/file1.rvpkg /path/to/file2.rvpkg
```

You can add multiple packages at the same time.

Remember that adding a package makes it become available for installation, it does not install it.

6.1.5 Removing a Package from a Support Area

```
shell> rvpkg -remove /path/to/area/Packages/file1.rvpkg
```

Unlike adding, the package in this case is the one in the support area's Packages directory. You can remove multiple packages at the same time.

If the package is installed rvpkg will interactively ask for confirmation to uninstall it first. You can override that by using -force as the first argument:

```
shell> rvpkg -force -remove /path/to/area/Packages/file1.rvpkg
```

6.1.6 Installing and Uninstalling Available Packages

```
shell> rvpkg -install /path/to/area/Packages/file1.rvpkg
shell> rvpkg -uninstall /path/to/area/Packages/file1.rvpkg
```

If files are missing when uninstalling rvpkg may complain. This can happen if multiple versions where somehow installed into the same area.

6.1.7 Combining Add and Install for Automated Installation

If you're using `rvpkg` from an automated installation script you will want to use the `-force` option to prevent the need for interaction. `rvpkg` will assume the answer to any questions it might ask is "yes". This will probably be the most common usage:

```
shell> rvpkg -force -install -add /path/to/area /path/to/some/file1.rvpkg
```

Multiple packages can be specified with this command. All of the packages are installed into `/path/to/area`. To force uninstall followed by removal:

```
shell> rvpkg -force -remove /path/to/area/Packages/file1.rvpkg
```

The `-uninstall` option is unnecessary in this case.

6.1.8 Overrideing Default Optional Package Load Behavior

If you want optional packages to be loaded by default for all users, you can do the following:

```
shell> rvpkg -optin /path/to/area/Packages/file1.rvpkg
```

In this case, rvkpg will rewrite the rvload2 file associated with the support area to indicate the package is no longer optional. The user can still unload the package if they want, but it will be loaded by default after running the command.

6.2 Package File Contents

A package file is zip file with at least one special file called `PACKAGE` along with `.mu`, `.so`, `.dylib`, and support files (plain text, images, icons, etc) which implement the actual package.

Creating a package requires the `zip` binary. The `zip` binary is usually part of the default install on each of the OSes that RV runs on.

When a package is installed, RV will place all of its contents into subdirectories in one of the `RV_SUPPORT_PATH` locations. If the `RV_SUPPORT_PATH` is not defined in the environment, it is assumed to have the value of `RV_HOME/plugins` followed by the home directory support area (which varies with each OS: see the user manual for more info). Files contained in one zip file will all be under the same support path directory; they will not be installed distributed over more than one support path location.

The install locations of files in the zip file is described in a filed called `PACKAGE` which must be present in the zip file. The minimum package file contains two files: `PACKAGE` and one other file that will be installed. A package zip file must reside in the subdirectory called `Packages` in one of the support path locations in order to be installed. When the user adds a package in the RV package manager, this is where the file is copied to.

6.3 `PACKAGE` Format

The `PACKAGE` file is a YAML file (See http://www.yaml.org/) providing information about how the package is used and installed as well as user documentation. Every package must have a `PACKAGE` file with an accurate description of its contents.

The top level of the file may contain the following fields:

Field	Value Type	Required	Description
package	string	•	The name of the package in human readable form
author	string		The name of the author/creator of the package
organization	string		The name of the organization (company) the author created the package for
contact	email address		The email contact of the author/support person
version	version number	•	The package version
url	URL		Web location for the package where updates, additional documentation resides
rv	version number	•	The minimum version of RV which this package is compatible with
requires	zip file name list		Any other packages (as zip file names) which are required in order to install/load this package
icon	PNG file name		The name of an file with an icon for this package
imageio	file list		List of files in package which implement Image I/O
movieio	file list		List of files in package which implement Movie I/O
hidden	boolean		Either ”true” or ”false” indicating whether package should be visible by default in the package manager
system	boolean		Either ”true” or ”false” indicating whether the package was pre-installed with RV and cannot be removed/uninstalled
optional	boolean		Either ”true” or ”false” indicating whether the package should appear loaded by default. If true the package is not loaded by default after it is installed. Typically this is used only for packages that are pre-installed. (Added in 3.10.9)
modes	YAML list		List of modes implemented in the package
files	YAML list		List non-mode file handling information
description	HTML 1.0 string	•	HTML documentation of the package for user viewing in the package manager

Table 6.2: Top level fields of `PACKAGE` file.

Each element of the modes list describes one Mu module which is implemented as either a `.mu` file or a `.so` file. Files implementing modes are assumed to be Mu module files and will be placed in the `Mu` subdirectory of the support path location. The other fields are used to optionally create a menu item and/or a short cut key either of which will toggle the mode on/off. The load field indicates when the mode should be loaded: if the value is ”lazy” the mode will be loaded the first time it is activated, if the value is ”immediate” the mode will be loaded on start up.

Field	Value Type	Required	Description
file	string	•	The name of the file which implements the mode
menu	string		If defined, the string which will appear in a menu item to indicate the status (on/off) of the mode
shortcut	string		If defined and menu is defined the shortcut for the menu item
event	string		Optional event name used to toggle mode on/off
load	string	•	Either immediate or delay indicating when the mode should be loaded
icon	PNG image file		Icon representing the mode
requires	mode file name list		Names of other mode files required to be active for this mode to be active

Table 6.3: Mode Fields

As an example, the package `window_title-1.0.rvpkg` has a relatively simple `PACKAGE` file shown in listing 6.1.

```
package: Window Title
author: Tweak Software
organization: Tweak Software
contact: some email address of the usual form
version: 1.0
url: http://www.tweaksoftware.com
rv: 3.6
requires: ''

modes:
  - file: window_title
    load: immediate

description: |

  <p> This package sets the window title to something that indicates the
  currently viewed media.
  </p>

  <h2>How It Works</h2>

  <p> The events play-start, play-stop, and frame-changed, are bound to
  functions which call setWindowTitle(). </p>
```

Listing 6.1: PACKAGE File

When the package zip file contains additional support files (which are not specified as modes) the package manager will try to install them in locations according to the file type. However, you can also directly specify where the additional files go relative to the support path root directory.

Field	Value Type	Required	Description
file	string	•	The name of the file in the package zip file
location	string	•	Location to install file in relative to the support path root. This can contain the variable $PACKAGE to specify special package directories. E.g. SupportFiles/$PACKAGE is the support directory for the package.

Table 6.4: File Fields

For example if you package contains icon files for user interface, they can be forced into the support files area of the package like this:

```
files:
  - file: myicon.tif
    location: SupportFiles/$PACKAGE
```

Listing 6.2: Specify Auxillary File Location

6.4 Package Management Configuration Files

There are two files which the package manager creates and uses: `rvload2` (previous releases had a file called `rvload`) in the `Mu` subdirectory and `rvinstall` in the `Packages` subdirectory. `rvload2` is used on start up to load package modes and create stubs in menus or events for toggling the modes on/off if they are lazy loaded. `rvinstall` lists the currently known package zip files with a possible an asterix in front of each file that is installed. The `rvinstall` file in used only by the package manager in the preferences to keep track of which packages are which.

The `rvload2` file has a one line entry for each mode that it knows about. This file is automatically generated by the package manager when the user installs a package with modes in it. The first line of the file indicates the version number of the `rvload2` file itself (so we can change it in the future) followed by the one line descriptions.

For example, this is the contents of `rvload2` after installing the window title package:

```
3
window_title,window_title.zip,nil,nil,nil,true,true,false
```

The fields are:

1. The mode name (as it appears in a require statement in Mu)
2. The name of the package zip file the mode originally comes from
3. An optional menu item name
4. An optional menu shortcut/accelerator if the menu item exists
5. An optional event to bind mode toggling to
6. A boolean indicating whether the mode should be loaded immediately or not
7. A boolean indicating whether the mode should be activated immediately
8. A boolean indicating whether the mode is optional so it should not be loaded by default unless the user opts-in.[1]

Each field is separated by a comma and there should be no extra whitespace on the line. The `rvinstall` file is much simpler: it contains a single zip file name on each line and an asterix next to any file which is current known to be installed. For example:

```
crop.zip
layer_select.zip
metadata_info.zip
sequence_from_file.zip
*window_title.zip
```

In this case, five modes would appear in the package manager UI, but only the window title package is actually installed. The zip files should exist in the same directory that rvinstall lives in.

6.5 Developing a New Package

In order to start a new package there is a chicken and egg problem which needs to be overcome: the package system wants to have a package file to install.

The best way to start is to create a source directory somewhere (like your source code repository) where you can build the zip file form its contents. Create a file called `PACKAGE` in that directory by copying and pasting from either this manual (listing 6.1) or from another package you know works and edit the file to reflect what you will be doing (i.e. give it a name, etc).

[1] Added in 3.10.9. The rvload2 file version was also bumped up to version 3.

If you are writing a Mu module implementing a mode or widget (which is also a mode) then create the .mu file in that directory also.

You can at that point use zip to create the package like so:

```
shell> zip new_package-0.0.rvpkg PACKAGE the_new_mode.mu
```

This will create the `new_package-0.0.rvpkg` file. At this point you're ready to install your package that doesn't do anything. Open RV's preferences and in the package manager UI add the zip file and install it (preferably in your home directory so its visible only to you while you implement it).

Once you've done this, the `rvload2` and `rvinstall` files will have been either created or updated automatically. You can then start hacking on the installed version of your Mu file (not the one in the directory you created the zip file in). Once you have it working the way you want copy it back to your source directory and create the final zip file for distribution and delete the one that was added by RV into the `Packages` directory.

6.5.1 Older Package Files (.zip)

RV version 3.6 used the extension `.zip` for its package files. This still works, but newer versions prefer the extension `.rvpkg` along with a preceeding version indicator. So a new style package will look like: `rvpackagename-X.Y.rvpkg` where `X.Y` is the package version number that appears in the `PACKAGE` file. New style package files are required to have the version in the file name.

6.5.2 Using the Mode Manager While Developing

Its possible to delay making an actual package file when starting developement on individual modes. You can force RV to load your mode (assuming its in the `MU_MODULE_PATH` someplace) like so:

```
shell> rv -flags ModeManagerLoad=my_new_mode
```

where `my_new_mode` is the name of the `.mu` file with the mode in it (without the extension).

You can get verbose information on what's being loaded and why (or why not by setting the verbose flag):

```
shell> rv -flags ModeManagerVerbose
```

The flags can be combined on the command line.

```
shell> rv -flags ModeManagerVerbose ModeManagerLoad=my_new_mode
```

If your package is installed already and you want to force it to be loaded (this overrides the user preferences) then:

```
shell> rv -flags ModeManagerPreload=my_already_installed_mode
```

similarily, if you want to force a mode not to be loaded:

```
shell> rv -flags ModeManagerReject=my_already_installed_mode
```

6.5.3 Using `-debug mu`

Normally, RV will compile Mu files to conserve space in memory. Unfortunately, that means loosing a lot of information like source locations when exceptions are thrown. You can tell RV to allow debugging information by adding `-debug mu` to the end of the RV command line. This will consume more memory but report source file information when displaying a stack trace.

6.5.4 The Mu API Documentation Browser

The Mu modules are documented dynamically by the documentation browser. This is available under RV's help menu ”Mu API Documentation Browser”.

6.6 Loading Versus Installing and User Override

The package manager allows eash user to individually install and uninstall packages in support directories that they have permission in. For directories that the user does not have permission in the package manager maintains a separate list of packages which can be excluded by the user.

For example, there may be a package installed facility wide owned by an administrator. The support directory with facility wide packages only allows read permission for normal users. Packages that were installed and loaded by the administrator will be automatically loaded by all users.

In order to allow a user to override the loading of system packages, the package manager keeps a list of packages not to load. This is kept in the user's preferences file (see user manual for location details). In the package manager UI the ”load” column indicates the user status for loading each package in his/her path.

6.6.1 Optional Packages

The load status of optional packages are also kept in the user's preferences, however these packages use a different preference variable to determine whether or not they should be loaded. By default optional packages are not loaded when installed. A package is made optional by setting the “optional” value in the `PACKAGE` file to true.

Chapter 7

A Simple Package

This first example will show how to create a package that defines some key bindings and creates a custom personal menu. You will not need to edit a `.rvrc.mu` file to do this as in previous versions.

We'll be creating a package intended to keep all our personal customizations. To start with we'll need to make a Mu module that implements a new mode. At first won't do anything at all: just load at start up. Put the following in to a file called `mystuff.mu`.

```
use rvtypes;
use extra_commands;
use commands;

module: mystuff {

class: MyStuffMode : MinorMode
{
    method: MyStuffMode (MyStuffMode;)
    {
        init("mystuff-mode",
             nil,
             nil,
             nil);
    }
}

\: createMode (Mode;)
{
    return MyStuffMode();
}

} // end module
```

Now we need to create a `PACKAGE` file in the same directory before we can create the package zip file. It should look like this:

Assuming both files are in the same directory, we create the zip file using this command from the shell:

```
shell> zip mystuff-1.0.rvpkg PACKAGE mystuff.mu
```

The file `mystuff-1.0.rvpkg` should have been created. Now start RV, open the preferences package pane and add the `mystuff-1.0.rvpkg` package. You should now be able to install it. Make sure the package is both installed and loaded in your home directory's RV support directory so its private to you.

At this point, we'll edit the installed Mu file directly so we can see results faster. When we have something we like, we'll copy it back to the original mystuff.mu and make the rvpkg file again with the new code. Be

```
package: My Stuff
author: M. VFX Artiste
version: 1.0
rv: 3.6
requires: ''

modes:
  - file: mystuff
    load: immediate

description: |
  <p>M. VFX Artiste's Personal RV Customizations</p>
```

careful not to uninstall the mystuff package while we're working on it or our changes will be lost. Alternately, for the more paranoid (and wiser), we could edit the file elsewhere and simply copy it onto the installed file.

To start with let's add two functions on the "<" and ">" keys to speed up and slow down the playback by increasing and decreasing the FPS. There are two main this we need to do: add two method to the class which implement speeding up and slowing down, and bind those functions to the keys.

First let's add the new methods after the class constructor `MyStuffMode()` along with two global bindings to the "<" and ">" keys. The class definition should now look like this:

```
...

class: MyStuffMode : MinorMode
{
    method: MyStuffMode (MyStuffMode;)
    {
        init("mystuff-mode",
             [("key-down-->", faster, "speed up fps"),
               ("key-down--<", slower, "slow down fps")],
             nil,
             nil);
    }

    method: faster (void; Event event)
    {
        setFPS(fps() * 1.5);
        displayFeedback("%g fps" % fps());
    }

    method: slower (void; Event event)
    {
        setFPS(fps() * 1.0/1.5);
        displayFeedback("%g fps" % fps());
    }
}
```

The bindings are created by passing a list of tuples to the init function. Each tuple contains three elements: the event name to bind to, the function to call when it is activated, and a single line description of what it does. In Mu a tuple is formed by putting parenthesis around comma separated elements. A list is formed by enclosing its elements in square brackets. So a list of tuples will have the form:

```
[ (...), (...), ... ]
```

Where the "..." means "and so on". The first tuple in our list of bindings is:

```
(key-down-->, faster, speed up fps)
```

So the event in this case is `key-down-->` which means the point at which the > key is pressed. The symbol `faster` is refering to the method we declared at line 14. So faster will be called whenever the key is pressed. Similarily we bind `slower` (from line 20) to `key-down--<`.

```
("key-down--<", slower, "slow down fps")
```

And to put them in a list requires enclose the two of them in square brackets:

```
[("key-down-->", faster, "speed up fps"),
 ("key-down--<", slower, "slow down fps")]
```

To add more bindings you create more methods to bind and add additional tuples to the list.

The python version of above looks like this:

```
from rv.rvtypes import *
from rv.commands import *
from rv.extra_commands import *

class PyMyStuffMode(MinorMode):

   def __init__(self):
        MinorMode.__init__(self)
        self.init("py-mystuff-mode",
                    [ ("key-down-->", self.faster, "speed up fps"),
                      ("key-down--<", self.slower, "slow down fps") ],
                    None,
                    None)

    def faster(self, event):
        setFPS(fps() * 1.5)
        displayFeedback("%g fps" % fps(), 2.0);

    def slower(self, event):
        setFPS(fps() * 1.0/1.5)
        displayFeedback("%g fps" % fps(), 2.0);


def createMode():
    return PyMyStuffMode()
```

7.1 How Menus Work

Adding a menu is fairly straightforward if you understand how to create a `MenuItem`. There are different types of `MenuItems`: items that you can select in the menu and cause something to happen, or items that are themselves menus (sub-menu). The first type is constructed using this constructor (shown here in prototype form) for Mu:

```
MenuItem(string       label,
         (void;Event) actionHook,
         string       key,
         (int;)       stateHook);
```

or in Python this is specified as a tuple:

```
("label", actionHook, "key", stateHook)
```

The `actionHook` and `stateHook` arguments need some explaination. The other two (the `label` and `key`) are easier: the `label` is the text that appears in the menu item and the `key` is a hot key for the menu item.

The `actionHook` is the purpose of the menu item–it is a function or method which will be called when the menu item is activated. This is just like the method we used with `bind()` — it takes an `Event` object. If `actionHook` is `nil`, than the menu item won't do anything when the user selects it.

The `stateHook` provides a way to check whether the menu item should be enabled (or greyed out)–it is a function or method that returns an `int`. In fact, it is really returning one of the following symbolic constants: `NeutralMenuState`, `UncheckMenuState`, `CheckedMenuState`, `MixedStateMenuState`, or `DisabledMenuState`. If the value of `stateHook` is `nil`, the menu item is assumed to always be enabled, but not checked or in any other state.

A sub-menu `MenuItem` can be create using this constructor in Mu:

```
MenuItem(string     label,
         MenuItem[] subMenu);
```

or a tuple of two elements in Python:

```
("label", subMenu)
```

The `subMenu` is an array of `MenuItems` in Mu or a list of menu item tuples in Python.

Usually we'll be defining a whole menu — which is an array of `MenuItems`. So we can use the array initialization syntax to do something like this:

```
let myMenu = MenuItem {"My Menu", Menu {
    {"Menu Item", menuItemFunc, nil, menuItemState},
    {"Other Menu Item", menuItemFunc2, nil, menuItemState2}
}}
```

Finally you can create a sub-menu by nesting more `MenuItem` constructors in the `subMenu`.

```
MenuItem myMenu = {"My Menu", Menu {
        {"Menu Item", menuItemFunc, nil, menuItemState},
        {"Other Menu Item", menuItemFunc2, nil, menuItemState2},
        {"Sub-Menu", Menu {
             {"First Sub-Menu Item", submenuItemFunc1, nil, submenu1State}
        }}
    }};
```

in Python this looks like:

```
("My Menu", [
  ("Menu Item", menuItemFunc, None, menuItemState),
  ("Other Menu Item", menuItemFunc2, None, menuItemState2)])
```

You'll see this on a bigger scale in the rvui module where most the menu bar is declared in one large constructor call.

7.2 A Menu in MyStuffMode

Now back to our mode. Let's say we want to put our faster and slower functions on menu items in the menu bar. The fourth argument to the init() function in our constructor takes a menu representing the menu bar. You only define menus which you want to either modify or create. The contents of our main menu will be merged into the menu bar.

By *merge into* we mean that the menus with the same name will share their contents. So for example if we add the File menu in our mode, RV will not create a second File menu on the menu bar; it will add the contents of our File menu to the existing one. On the other hand if we call our menu MyStuff RV will create a brand new menu for us (since presumably MyStuff doesn't already exist). This algorithm is applied recursively so sub-menus with the same name will also be merged, and so on.

So let's add a new menu called MyStuff with two items in it to control the FPS. In this example, we're only showing the actual `init()` call from `mystuff.mu`:

```
init("mystuff-mode",
     [ ("key-down-->", faster, "speed up fps"),
       ("key-down--<", slower, "slow down fps") ],
     nil,
     Menu {
         {"MyStuff", Menu {
                 {"Increase FPS", faster, nil},
                 {"Decrease FPS", slower, nil}
             }
         }
     });
```

Normally RV will place the new menu (called "MyStuff") just before the `Windows` menu.

If we wanted to use menu accelerators instead of (or in addition to) the regular event bindings we add those in the menu item constructor. For example, if we wanted to also use the keys - and = for slower and faster we could do this:

```
init("mystuff-mode",
     [ ("key-down-->", faster, "speed up fps"),
       ("key-down--<", slower, "slow down fps") ],
     nil,
     Menu {
         {"MyStuff", Menu {
                 {"Increase FPS", faster, "="},
                 {"Decrease FPS", slower, "-"}
             }
         }
     });
```

The advantage of using the event bindings instead of the accelerator keys is that they can be overriden and mapped and unmapped by other modes and "chained" together. Of course we could also use > and < for the menu accelerator keys as well (or instead of using the event bindings).

The Python version of the script might look like this:

```
from rv.rvtypes import *
from rv.commands import *
from rv.extra_commands import *

class PyMyStuffMode(MinorMode):

   def __init__(self):
        MinorMode.__init__(self)
        self.init("py-mystuff-mode",
                  [ ("key-down-->", self.faster, "speed up fps"),
                    ("key-down--<", self.slower, "slow down fps") ],
                  None,
                  [ ("MyStuff",
                      [ ("Increase FPS", self.faster, "=", None),
                        ("Decrease FPS", self.slower, "-", None)] )] )

    def faster(self, event):
        setFPS(fps() * 1.5)
        displayFeedback("%g fps" % fps(), 2.0);

    def slower(self, event):
        setFPS(fps() * 1.0/1.5)
        displayFeedback("%g fps" % fps(), 2.0);


def createMode():
    return PyMyStuffMode()
```

7.3 Finishing up

Finally, we'll create the final rvpkg package by copying `mystuff.mu` back to our tempory directory with the `PACKAGES` file where we originally made the rvpkg file.

Next start RV and uninstall and remove the mystuff package so it no longer appears in the package manager UI. Once you've done this recreate the rvpkg file from scratch with the new mystuff.mu file and the `PACKAGES` file:

```
shell> zip mystuff-1.0.rvpkg PACKAGES mystuff.mu
```

or if you're using python:

```
shell> zip mystuff-1.0.rvpkg PACKAGES mystuff.py
```

You can now add the latest `mysuff-1.0.rvpkg` file back to RV and use it. In the future add personal customizations directly to this package and you'll always have a single file you can install to customize RV.

Chapter 8

The Custom Matte Package

Now that we've tried the simple stuff, let's do something useful. [1] RV has a number of settings for viewing mattes. These are basically regions of the frame that are darkened or completely blackened to simulate what an audience will see when the movie is projected. The size and shape of the matte is an artistic decision and sometimes a unique matte will be required.

You can find various common mattes already built into RV under the View menu.

In this example we'll create a package that reads a file when RV starts to get a list of matte geometry and names. We'll make a custom menu out of these which will set some state in the UI. We'll also use the mode render function to draw the mattes.

To start with, we'll assume that the path to the file containing the mattes is located in an environment variable called `RV_CUSTOM_MATTE_DEFINITIONS`. We'll get the value of that variable, open and parse the file, and create a data struct holding all of the information about the mattes.

8.1 Creating the Package

Use the same method described in Chapter 7 to begin working on the package. If you haven't read that chapter please do so first. A completed version of the package created in this chapter is included in the RV distribution. So using that as reference is a good idea.

8.2 The Custom Matte File

The file will be a very simple comma separated value (CSV) file. Each line will start with the name of the custom matte followed by four floating point values indicating the distances to each edge as a percentage of the frame width and height and some text to render under the matte line. So each line will look something like this:

```
Matte Name, left, right, bottom, top, text
```

8.3 Parsing the Matte File

Before we actually parse the file, we should decide what we want when we're done. In this case we're going to make our own data structure to hold the information in each line of the file. We'll call this a `MatteDescription`. Here's a simple way to define the type:

```
class: MatteDescription
{
    string name;
```

[1]Previous versions of this manual presented a different approach which still works in RV 3.6, but is no longer the preferred method.

```
    float left;
    float right;
    float bottom;
    float top;
    string text;
}
```

We don't need to define any special constructors because Mu supplies a few default ones. One of them is a function that takes one argument for each field. We'll be using that to create instances of `MatteDescription`.

Next we'll write a method[2] for our mode that does the parsing and returns a dynamic array of `MatteDescriptions`.

```
 1  method: parseMatteFile (MatteDescription[]; string filename)
 2  {
 3      use io;
 4
 5      if (!path.exists(filename)) return nil;
 6
 7      let file       = ifstream(filename),
 8          everything = read_all(file),
 9          lines      = everything.split("\n\r");
10
11      file.close();
12
13      MatteDescription[] mattes;
14
15      for_each (line; lines)
16      {
17          let tokens = line.split(",");
18
19          if (!tokens.empty())
20          {
21              mattes.push_back( MatteDescription(tokens[0],
22                                                 float(tokens[1]),
23                                                 float(tokens[2]),
24                                                 float(tokens[3]),
25                                                 float(tokens[4]),
26                                                 tokens[5]) );
27          }
28      }
29
30      return mattes;
31  }
```

Listing 8.1: Parsing the Custom Matte File

There are a number of things to note in this function. First of all, in order to do file I/O in Mu we need the `io` module. Line 3 loads the module and puts all of its symbols in the current namespace. We could have used `require` instead of `use`, but functions like the `ifstream` constructor would need to be written `io.ifstream(...)`.

In line 5 we check to see if the file actually exists and if not simply return `nil`. So the caller of this function will have to check if anything was parsed and do something intelligent if not.

The let section at line 7 declares three variables. The `file` variable is a newly created `ifstream` (input file stream). The next line reads the entire file as a string and assigns it to the variable `everything`. Finally, `everything` is split into an array of lines which is assigned to `lines`.

[2]If you are unfamiliar with object oriented programing you can substitute the word function for method. This manual will sometimes refer to a method as a function. It will never refer to a non-method function as a method.

At this point we don't need the file any more so its closed on line 11.

Line 13 creates an empty dynamic array of MatteDescription objects to be filled in.

The `for_each` loop iterates over the `lines` variable. Each time through the loop, the next element in `lines` is assigned to the variable `line`. The line is split over commas since that's how defined the fields of each line.

If there are no tokens after splitting the line, that means the line is empty and we ignore it. Otherwise, a new `MatteDescription` is created using the split line and appended to the end of the `mattes` array.

Finally, the function returns whatever `MatteDesriptions` it managed to parse.

But wait, we'll need something to read our environment variable. The easiest thing to do is make a method that gets the environment variable and calls our new parse function on the file it finds. We'll be calling this instead of our `parseMatteFile()` function. If we want to change how it looks for the file later, we can change it without touching our parsing function:

```
method: findAndParseMatteFile (MenuDescription[];)
{
    use system;
    return parseMatteFile( getenv("RV_CUSTOM_MATTE_DEFINITIONS") );
}
```

The method `findAndParseMatteFile()` is a one liner: just call our other function with the result of the environment variable look up. The `getenv()` function is in the `system` module so we had to `use` it.

We'll need two "use" lines first thing at the top of the module definition. These lines import the `system` and `io` modules and their namespaces. These imports are available to all functions in the module. So we no longer need to have them inside each function. To use this module in some other code you can either use `require` or `use`. The only difference is how you call the functions:

```
require customMattes;
let mattes = customMattes.findAndParseMatteFile();
// --or--
use customMattes;
let mattes = findAndParseMatteFile();
```

Note that `use` actually does two things: first it loads the module if its not already loaded (just like `require` does) and then it uses the module's namespace. The use of the namespace lasts through the end of the current scope (the next end curly brace). So if we wanted to, we could put the `use` line at the top of the file and it would be in effect for everything in the file.

Finally, we'll add some member variables to our mode to hold the MatteDescription objects we parsed.

At this point the `custom_mattes.mu` file looks like this:

```
module: custom_mattes {
use rvtypes;
use io;
use commands;
use extra_commands;
use system;

class: CustomMatteMinorMode : MinorMode
{
    class: MatteDescription
    {
        string name;
        float left;
        float right;
        float bottom;
        float top;
        string text;
    }
    MatteDescription    _currentMatte;
    MatteDescription[]  _mattes;

    method: parseMatteFile (MatteDescription[]; string filename)
    {
        ... see above ...
    }

    method: findAndParseMatteFile (MatteDescription[];)
    {
        ... see above ...
    }

    method: CustomMatteMinorMode (CustomMatteMinorMode;)
    {
        _mattes = findAndParseMatteFile();

        this.init("custom-mattes",
                  nil,
                  nil,
                  nil);

    }
}

\: createMode (Mode;)
{
    return CustomMatteMinorMode();
}

} // end module
```

8.4 The CustomMattesMode Constructor

The mode constructor needs to do three things: call the file parsing function, do something sensible if the matte file parsing fails, and build a menu with the items found in the matte file.

The parsing is simply calling `findAndParseMatteFile()`. If an error occurs, the function will throw. So we need to enclose it in a try block and report an error and continue if it fails.

```
method: CustomMatteMinorMode (CustomMatteMinorMode;)
{
    MenuItem matteMenu = nil;

    try
    {
        _mattes = findAndParseMatteFile();

        let matteItems = Menu();

        matteItems.push_back( ... menu item for no matte ...);

        for_each (m; _mattes)
        {
```

```
15              matteItems.push_back( ... menu item for matte ...);
16          }
17
18          if (!matteItems.empty())
19          {
20              matteMenu =
21                  MenuItem("View",
22                           Menu(MenuItem("_", nil),
23                                MenuItem("Custom Mattes",
24                                         matteItems)));
25          }
26
27      }
28      catch (...)
29      {
30          _mattes = nil;
31      }
32
33      init("Custom Matte",
34           nil,
35           nil,
36           Menu(matteMenu));
37  }
```

Listing 8.2: Mode Constructor Outline

Line 7 calls the parser function we created ealier. If the function fails, it will throw and control resume at line 28.

A menu array is created at line 9 which is subsequently filled with an option to draw nothing (line **??**). Each of the matte items read from the file are then added to the `Menu` in the `for_each` loop at line 13. Finally the View menu is created with a Custom Mattes sub-menu to hold the items (line 20)

In chapter 7 menus are created with methods to call when the user selects them. However, we have to do something a bit more complex in this example: each method needs to know about a particular matte to make it current, but we don't know about the mattes until the file is parsed. The solution is to somehow associate each call to a method with its matte. In this example we'll use a technique called *partial application*[3] to wrap each function call with its matte.

The `_currentMatte` variable in the mode indicates which matte will be rendered. The menu item method will need to set this to an incoming matte:

```
method: selectMatte (void; Event event, MatteDescription m)
{
    _currentMatte = m;
    redraw();
}
```

Notice that we didn't say *which* matte to set it to. The function just sets the value to whatever its argument is. Since this function is going to be called when the menu item is selected it needs to be an event function (a function which takes an `Event` as an argument and returns nothing). In the case where we want no matte drawn, we'll pass in `nil`.

The menu state function (which will put a check mark next to the current matte) has a similar problem. In this case we'll use a mechanism with similar results: a closure[4]. We'll create a method which returns a function given a matte. The returned function will be our menu state function. This sounds complicated, but its simple in use:

[3] One kind of partial application which may be illuminating is called currying and is described here: http://en.wikipedia.org/wiki/Currying

[4] http://en.wikipedia.org/wiki/Closure_(computer_science)

```
1  method: currentMatteState ((int;); MatteDescription m)
2  {
3      \: (int;)
4      {
5          return if this._currentMatte eq m
6                     then CheckedMenuState
7                     else UncheckedMenuState;
8      };
9  }
```

The thing to note here is that the parameter `m` (line 1) passed into `currentMatteState()` is being used inside the anonymous function that it returns. The `m` inside the anonymous function (line 5)is known as a *free variable*. The value of this variable at the time that `currentMatteState()` is called becomes wrapped up with the returned function. One way to think about this is that each time you call `currentMatteState()` with a new value for `m`, it will return a different anonymous function where the internal `m` (line 5) is replaced the value of `currentMatteState()`'s `m`.

So how do we build the menus? In listing 8.2 the for_each loop at line 13 will create a menu item and push it onto the menu. Here's how we'll do it:

```
1  for_each (m; _mattes)
2  {
3      matteItems.push_back( MenuItem( m.name,
4                                      selectMatte(,m),
5                                      nil,
6                                      currentMatteState(m)) );
7  }
```

The call to selectMatte() at line 4 performs the partial application we talked about above. Notice that the first argument to the function is missing. For the menu state function we call `currentMatteState()` at line 6 resulting in a unique anonymous function for our matte.

For the case with no matte we do the same thing, but use `nil` in place of a matte:

```
matteItems.push_back( MenuItem("No Matte",
                               selectMatte(,nil),
                               nil,
                               currentMatteState(nil)) );
```

So the full mode constructor function now looks like this:

```
method: CustomMatteMinorMode (CustomMatteMinorMode;)
{
    MenuItem matteMenu = nil;

    try
    {
        _mattes = findAndParseMatteFile();

        let matteItems = Menu();

        matteItems.push_back( MenuItem("No Matte",
                                       selectMatte(,nil),
                                       nil,
                                       currentMatteState(nil)) );

        for_each (m; _mattes)
        {
            matteItems.push_back(MenuItem( m.name,
                                           selectMatte(,m),
                                           nil,
                                           currentMatteState(m)) );
        }

        if (!matteItems.empty())
        {
            matteMenu = MenuItem("View",
                                Menu(MenuItem("_", nil),
                                     MenuItem("Custom Mattes", matteItems)));
        }

    }
    catch (...)
    {
        _mattes = nil;
    }

    init("Custom Matte",
         nil,
         nil,
         Menu(matteMenu));
}
```

8.5 Rendering the Matte

In addition to parsing the matte file, we also need to draw the mattes. The basic idea will be to grey or black out large sections of the underlying image.

```
method: render (void; Event event)
{
    if (_currentMatte eq nil) return;

    \: sort (Vec2[]; Vec2[] array)
    {
        // only handles flipping not flopping right now
        if (array[0].y < array[2].y)
           return Vec2[] {array[3], array[2], array[1], array[0]}
        else
               return array;
    }

    State state = data();
    setupProjection(event.domain().x, event.domain().y);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    gltext.size(20.0);
    gltext.color(Color(1,1,1,1) * .5);

    let {_, l, r, b, t, text} = _currentMatte;

    let bounds = gltext.bounds(text),
        th     = bounds[1] + bounds[3];

    Color c = state.config.matteColor;
    c.w = state.matteOpacity;

    glColor(c);

    for_each (ri; sourcesRendered())
    {
        let g = sort(sourceGeometry(ri.name)),
            w = g[2].x - g[0].x,
            h = g[2].y - g[0].y,
            a = w / h,
            x0 = g[0].x + l * w,
            x1 = g[2].x - r * w,
            y0 = g[0].y + t * h,
            y1 = g[2].y - b * h;

        glBegin(GL_QUADS);

        //  Top
        glVertex(g[0]);
        glVertex(g[1]);
        glVertex(g[1].x, y0);
        glVertex(g[0].x, y0);

        //  Bottom
        glVertex(g[3].x, y1);
        glVertex(g[2].x, y1);
        glVertex(g[2]);
        glVertex(g[3]);

        //  Left
        glVertex(g[0].x, y0);
        glVertex(x0, y0);
        glVertex(x0, y1);
        glVertex(g[0].x, y1);

        //  Right
        glVertex(g[1].x, y0);
        glVertex(x1, y0);
        glVertex(x1, y1);
        glVertex(g[1].x, y1);

        glEnd();

        gltext.writeAt(x0, y1 - th - 5, text);
    }

    glDisable(GL_BLEND);
}
```

There are a few things we haven't seen before in there. First of all, its getting some crucial information what was rendered. In this case we want to loop over all of the images which have been draw by the renderer. The function `sourcesRendered()` returns an array of type `RenderedSourceInfo`, one for each image.

In this case, we're only interested in the name of each source so we can call the command `sourceGeometry()` in line 35. `sourceGeometry()` returns a `Vec2[]` of the four corners of the image in the view space. Because not all images have the same orientation (origin on top or bottom, etc) the results are sorted in an order that makes drawing easiest. This is done by calling a small utility function `sort()` which is declared inside the `render()` method.

8.6 The Finished custom_mattes.mu File

```
module: custom_mattes {
use gl;
use glu;
use rvtypes;
use io;
use commands;
use extra_commands;
use glyph;
use system;

class: CustomMatteMinorMode : MinorMode
{
    class: MatteDescription
    {
        string name;
        float left;
        float right;
        float bottom;
        float top;
        string text;
    }

    MatteDescription    _currentMatte;
    MatteDescription[]  _mattes;

    method: parseMatteFile (MatteDescription[]; string filename)
    {
        if (!path.exists(filename)) return nil;

        let file       = ifstream(filename),
            everything = read_all(file),
            lines      = everything.split("\n\r");

        file.close();

        MatteDescription[] mattes;

        for_each (line; lines)
        {
            let tokens = line.split(",");

            if (!tokens.empty())
            {
                //
                //  This will throw if we don't have enough tokens.  So the
                //  called of the parseMatteFile() function should make sure
                //  to catch.
                //

                mattes.push_back( MatteDescription(tokens[0],
                                                   float(tokens[1]),
                                                   float(tokens[2]),
                                                   float(tokens[3]),
                                                   float(tokens[4]),
                                                   tokens[5]) );
            }
        }

        return mattes;
    }

    method: findAndParseMatteFile (MatteDescription[];)
    {
        return parseMatteFile( getenv("RV_CUSTOM_MATTE_DEFINITIONS") );
    }

    method: selectMatte (void; Event event, MatteDescription m)
    {
        _currentMatte = m;
        redraw();
    }

    method: currentMatteState ((int;); MatteDescription m)
    {
        \: (int;)
        {
            return if this._currentMatte eq m
                      then CheckedMenuState
                      else UncheckedMenuState;
        };
    }

    method: CustomMatteMinorMode (CustomMatteMinorMode;)
    {
        MenuItem matteMenu = nil;

        try
        {
            //_mattes = findAndParseMatteFile();
            _mattes = parseMatteFile("/Users/jimh/mattes");

            let matteItems = Menu();

            matteItems.push_back( MenuItem("No Matte",
                                           selectMatte(,nil),
                                           nil,
                                           currentMatteState(nil)) );

            for_each (m; _mattes)
            {
                matteItems.push_back( MenuItem( m.name,
                                                selectMatte(,m),
                                                nil,
                                                currentMatteState(m)) );
            }

            if (!matteItems.empty())
            {
                matteMenu = MenuItem("View",
                                     Menu(MenuItem("_", nil),
                                          MenuItem("Custom Mattes",
                                                   matteItems) ) );
            }

        }
        catch (...)
        {
            _mattes = nil;
        }

        init("Custom Matte",
             nil,
             nil,
             Menu(matteMenu));
    }

    method: render (void; Event event)
    {
        if (_currentMatte eq nil) return;

        \: sort (Vec2[]; Vec2[] array)
        {
            // only handles flipping not flopping right now
            if array[0].y < array[2].y
                then Vec2[] { array[3], array[2], array[1], array[0] }
                else array;
        }


        State state = data();
        setupProjection(event.domain().x, event.domain().y);

        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        gltext.size(20.0);
        gltext.color(Color(1,1,1,1) * .5);

        let {_, l, r, b, t, text} = _currentMatte;
        let bounds = gltext.bounds(text),
            th     = bounds[1] + bounds[3];

        Color c = state.config.matteColor;
        c.w = state.matteOpacity;

        glColor(c);

        for_each (ri; sourcesRendered())
        {
            let g = sort(sourceGeometry(ri.name)),
                w = g[2].x - g[0].x,
                h = g[2].y - g[0].y,
                a = w / h,
                x0 = g[0].x + l * w,
                x1 = g[2].x - r * w,
                y0 = g[0].y + t * h,
                y1 = g[2].y - b * h;

            glBegin(GL_QUADS);

            //  Top
            glVertex(g[0]);
            glVertex(g[1]);
            glVertex(g[1].x, y0);
            glVertex(g[0].x, y0);

            //  Bottom
            glVertex(g[3].x, y1);
            glVertex(g[2].x, y1);
            glVertex(g[2]);
            glVertex(g[3]);

            //  Left
            glVertex(g[0].x, y0);
            glVertex(x0, y0);
            glVertex(x0, y1);
            glVertex(g[0].x, y1);

            //  Right
            glVertex(g[1].x, y0);
            glVertex(x1, y0);
            glVertex(x1, y1);
            glVertex(g[1].x, y1);

            glEnd();

            gltext.writeAt(x0, y1 - th - 5, text);
        }

        glDisable(GL_BLEND);
    }
}

\: createMode (Mode;)
{
    return CustomMatteMinorMode();
}

} // end module
```

Chapter 9

Automated Color and Viewing Management

Color management in RV can be broken into three separate issues:

- Determination of the input color space
- Deciding whether the input color space should be converted to the linear working space and with what transform
- Displaying the working color space on a particular device possibly in a manner which simulates another device (e.g, film look on an LCD monitor).

Each of the above corresponds to a set of features in RV which can be automated:

- Examining particular image attributes its often possible to determine color space. Some images may use a naming convention or may be located in a particular place on a file system which indicates its color space. Its even possible that a separate file or program needs to be executed to get the actual color space.
- Input spaces can be transformed to working spaces using the built in transforms like sRGB, gamma, or Kodak log space to linear space. If RV does not have a built-in transform for the color space, a file LUT (one per input source) may be used to interpolate independant channel functions using a channel LUT or a general function of `R` `G` and `B` channels using a 3D LUT.
- Ideally RV will have a fixed set of display transform which map a linear space to a display. This makes it possible to load multiple sets of images with differing color spaces, transform them to a common linear working space, and display them using a global display transform. RV has built-in transform for sRGB and gamma and can also use a channel or 3D LUT if a custom function is needed.

In addition to the color issues there are a few others which might need to be detected and/or corrected:

- Unrecorded or incorrect pixel aspect ratios (e.g., DPX files with inaccurate headers)
- Special mattes which should be used with particular images
- Incorrect frame numbers
- Incorrect fps
- Specific production information which is not located in the image (e.g., shot information, tracking information)

RV lets you customize all of the above for your facility and workflow by hooking into the user interface code. The most important method of doing so is using special events generated by RV internally and setting internal state at that time.

9.1 The `new-source` Event

The new-source event is generated whenever a file is first added to a session. By binding a function to this event, its possible to configure any color space or other image dependant aspects of RV at the time the file is added. This can save a considerable amount of time and headache when a large nunber of people are using RV in differing circumstances.

See the sections below for information about creating a package which binds new-source to do color management.

9.2 The default `new-source` behavior

By default RV binds its own color management function located in the `source_setup.mu` file called `sourceSetup()`. This is part of the `source_setup` system package introduced in version 3.10.

Its a good idea to override or augment this package for use in production environments. For example, you may want to have certain default color behavior for technical directors using movie files which differs from how a coordinator might view them (the coordinator may be looking at movies in sRGB space instead of with a film simulation for example).

RV's default color management package does not account for the user, but it does try to use good defaults for incoming file formats. Here's the complete behavior shown as a set of heuristics applied in order:

1. If the incoming image is a TIFF file and it has no color space attribute assume its linear
2. If the image is JPEG or a quicktime movie file (.mov) and there is no color space attribute assume its in sRGB space
3. If there is an embedded ICC profile and that profile is for sRGB space use RV's internal sRGB space transform instead (because RV does not yet handle embedded ICC profiles)
4. If the image is TIFF and it was created by ifftoany, assume the pixel aspect ratio is incorrect and fix it
5. If the image is JPEG, has no pixel aspect ratio attribute and no densisty attribute and looks like it comes from Maya, fix the pixel aspect ratio
6. Use the proper built-in conversion for the color space indicated in the color space attribute of the image
7. Use the sRGB display transform if any color space was successfully determined for the input image(s)

From the user's point of view, the following situations will occur:

- A DPX or Cineon is loaded which is determined to be in Log space — turn on the built in log to linear converter
- A JPEG or Quicktime movie file is determined to be in sRGB space or if no space is specified assumed to be in sRGB space — apply the built-in sRGB to linear converter
- An EXR is loaded — assume its linear
- A TIFF file with no color space indication is assumed to be linear, if it does have a color space use that.
- A PNG file with no color space is assumed linear, otherwise use the color space attribute in the file
- Any file with a pixel aspect ratio attribute will be assumed to be correct (unless its determined to have come from Maya)
- The monitor's "gamma" will be accounted for automatically (because RV assumes the monitor is an sRGB device)

9.3 Breakdown of `sourceSetup()` in the `source_setup` Package

The `source_setup` system package defines the default `sourceSetup()` function. This is where RV's default color management comes from. The function starts by parsing the event contents (which contains the name of the file, the type of source node, and the source node name) as well as setting up the regular expressions used later in the function:[1]

```
let args      = event.contents().split(";;"),
    source    = args[0],
    colorNode = associatedNode("#RVColor", source),
    tformNode = associatedNode("#RVTransform2D", source),
    type      = args[1],
    file      = args[2],
    ext       = io.path.extension(file),
    dpxRE     = regex("dpx|DPX"),
    exrRE     = regex("((s|e)xr)|((S|E)XR)"),
    jpgRE     = regex("jpe?g|JPE?G"),
    tifRE     = regex("tiff?|TIFF?"),
    movRE     = regex("mov|MOV|avi|AVI|mp4|MP4");
```

The event.contents() function returns a string which might look something like this:

```
source000;;RVSource;;/path/to/file.mov
```

The `split()` function is used to create a dynamic array of strings which are then assigned to `source`, `type`, and `file`. The source node name is then used to find associated nodes (the color and transform nodes associated with the source node). The regular expressions are used to identify the file type from its extension.

The next section of the function iterates over the image attributes and caches the ones we're interested in. The most important of these is teh Colorspace attribute which is set by the file readers when the image color space is known.

```
for_each (a; sourceAttributes(source, file))
{
    let (name, val) = a;

    if      (name == "Colorspace/ICC Profile Name") ICCProfileName = val;
    else if (name == "Colorspace") Colorspace = val;
    else if (name == "JPEG/PixelAspect") JPEGPixelAspect = val;
    else if (name == "JPEG/Density") JPEGDensity = val;
    else if (name == "TIFF/ImageDescription") TIFFImageDescription = val;
}
```

The function `sourceAttributes()` returns the image attributes for a given file in a source. In this case we're passing in the source and file which caused the event. The return value of the function is a dynamic array of tuples of type `(string,string)` where the first element is the name of the attribute and the second is a string representation of the value. Each iteration through the loop, the next tuple is assign to the variable `a` and this is then used to assign `name` and `val`.

The variables `ICCProfileName`, `Colorspace`, `JPEGPixelAspect`, etc, are all variable of type `string` which are defined earlier in the function.

Before getting to the meat of the function, there are two helper functions declared: `setPixelAspect()` and `setFileColorSpace()`. These are declared directly in the scope of the `sourceSetup()` function so they may only be used inside of it. In addition these functions reference variables in the enclosing scope for convenience.

The next major section of the function matches the file name against the regular expressions that were declared at the beginning and against the values of some of the attribtues that were cached.

```
if (tifRE.match(file) && Colorspace == "")
{
    Colorspace = "Linear";
}
```

[1]The actual `sourceSetup()` function in `source_setup.mu` may differ from what is described here since it is constantly being refined.

```
else if ((jpgRE.match(file) || movRE.match(file)) && Colorspace == "")
{
    Colorspace = "sRGB";
}

if (ICCProfileName != "")
{
    if (regex.match("sRGB", ICCProfileName)) Colorspace = "sRGB";
}

if (TIFFImageDescription == "Image converted using ifftoany")
{
    setPixelAspect(1);
}

if (JPEGPixelAspect != "" && JPEGDensity != "")
{
    let (w, h, bits, ch, flt, pl) = sourceImageStructure(source, file);

    let attrPA   = float(JPEGPixelAspect),
        imagePA  = float(w) / float(h),
        testDiff = attrPA - 1.0 / imagePA;

    if (math.abs(testDiff) < .0001)
    {
        setPixelAspect(1);
    }
}
```

At this point in the function the color space of the input image will be known or assumed to be linear. Finally, we try to set the color space (which will result in the image pixels being converted to the linear working space). If this succeeds, use sRGB display as the default.

```
if (setFileColorSpace(Colorspace))
{
    setIntProperty("display.color.sRGB", int[] {1});
}
```

9.4 Setting up 3D and Channel LUTs

The default new-source event function does not set up any non-built-in transforms. When you need to automatically apply a LUT, as a file, look, or a display LUT, you need to do the following:

```
readLUT(file, nodeName);
setIntProperty(%s.lut.active" % nodeName, int[] {1});
updateLUT();
```

The nodeName will be "display" (or "#RVDisplayColor" to refer to it by type) for the display LUT. For a file or look LUT, you use the associated node name for the color node — in the default sourceSetup() function this would be the colorNode variable. The file parameter to readLUT() will be the name of the LUT file on disk and can be any of the LUT types that RV reads.

9.5 Building a Package For Color Management

As of RV 3.6 the recommend way to handle all event bindings is via a package. In version 3.10 the color management was made a system package. To customize color management you can either create a new package from scratch as described here, or copy, rename, and hack the existing source_setup package.

The use of new-source is no different from any other event. By creating a package you can override the existing behavior or modify it. It also makes it possible to have layers of color management packages which (assuming they don't contradict each other) can collectively create a desired behavior.

```
 1  module: custom_color_management {
 2  use rvtypes;
 3  use commands;
 4  use extra_commands;
 5  use system;
 6  use source_setup;
 7
 8  class: CustomColorManagementMode : MinorMode
 9  {
10      method: newSource (void; Event event)
11      {
12          // do work on the new source here
13          event.reject();
14      }
15
16      method: CustomColorManagementMode (CustomColorManagementMode;)
17      {
18          init("Custom Color Management",
19               nil,
20               [("new-source", newSource, "Color Setup")],
21               nil);
22      }
23  }
24
25  \: createMode (Mode;)
26  {
27      return CustomColorManagementMode();
28  }
29
30  } // end module
```

There are two things to note in here: the first is that pass our event binding in as the third argument of `init()` at line 20 as opposed to the second. The second argument binds events in the global event table whereas the third argument uses the local mode table. This keeps the existing global binding available.

The second thing to note is that our `newSource()` method has two choices: it can eat the event so no other function bound to it can process it or it can allow it to continue. To allow other functions bound to `new-source` to get the event the `reject()` function of the event must be called in `newSource()` as seen at line 13. Unless our mode is intended to completely override the behavior of the default source setup function its probably best to reject the event and allow others to process it.

The example code above for the custom_color_management package is included with RV.

Chapter 10

Network Communication

RV can communicate with multiple external programs via its network protocol. The mechanism is designed to function like a ”chat” client. Once a connection is established, messages can be sent and received including arbitrary binary data.

There are a number of applications which this enables:

- **Controlling RV remotely.** E.g., a program which takes input from a dial and button board or a mobile device and converts the input into commands to start/stop playback or scrubbing in RV.
- **Synchronizing RV sessions across a network**. This is how RV's sync mode is implemented: each RV serves as a controller for the other.
- **Monitoring a Running RV**. For VFX theater dailies the RV session driving the dailies could be monitored by an external program. This program could then indicate to others in the facility when their shots are coming up.
- **A Display Driver for a Renderer**. Renders like Pixar's RenderMan have a plug-in called a display driver which is normally used to write out rendered frames as files. Frequently this type of plug-in is also used to send pixels to an external frame buffer (like RV) to monitor the renderer's progress in real time. Its possible to write a display driver that talks to RV using the network protocol and send it pixels as they are rendered. A more advanced version might receive feedback from RV (e.g. a selected rectangle on the image) in order to recommend areas the renderer should render sooner.

Any number of network connections can be estabilished simultaneously, so for example its possible to have a synchronized RV session with a remote RV and drive it with an external hardware device at the same time.

10.1 Example Code

There are two working examples that come with RV: the rvshell program and pyNetwork.py python example.

The rvshell program uses a C++ library included with the distribution called TwkQtChat which you can use to make interfacing easier — especially if your program will use Qt. We highly recommend using this library since this is code which RV uses internally so it will always be up-to-date. The library is only dependent on the QtCore and QtNetwork modules.

The pyNetwork example implements the network protocol using only python native code. You can use it directly in python programs.

10.1.1 Using rvshell

To use rvshell, start RV from the command line with the network started and a default port of 45000 (to make sure it doesn't interfere with existing RV sessions):

```
shell> rv -network -networkPort 45000
```

Next start the rvshell program program from a different shell:

```
shell> rvshell user localhost 45000
```

Assuming all went well, this will start rvshell connected to the running RV. There are three things you can experiment with using rvhell: a very simple controller interface, a script editor to send portions of script or messages to RV manually, and a display driver simulator that sends stereo frames to RV.

Start by loading a sequence of images or a quicktime movie into RV. In rvshell switch to the ”Playback Control” tab. You should be able to play, stop, change frames and toggle full screen mode using the buttons on the interface. This example sends simple Mu commands to RV to control it. The feedback section of the interface shows the RETURN message send back from RV. This shows whatever result was obtained from the command.

The ”Raw Event” section of the interface lets you assemble event messages to send to RV manually. The default event message type is `remote-eval` which will cause the message data to be treated like a Mu script to execute. There is also a `remote-pyeval` event which does the same with Python (in which case you should type in Python code instead of Mu code). Messages sent this way to RV are translated into UI events. In order for the interface code to respond to the event something must have bound a function to the event type. By default RV can handle `remote-eval` and `remote-pyeval` events, but you can add new ones yourself.

When RV receieves a `remote-eval` event it executes the code and looks for a return value. If a return value exists, it converts it to a string and sends it back. So using `remote-eval` its possible to querry RV's current state. For example if you load an image into RV and then send it the command `renderedImages()` it will return a Mu struct as a string with information about the rendered image. Similarily, sending a `remote-pyeval` with the same command will return a Python dictionary as a string with the same information.

The last tab ”Pixels” can be used to emulate a display driver. Load a JPEG image into rvshell's viewer (don't try something over 2k — rvshell is using Qt's image reader). Set the number of tiles you want to send in X and Y, for example 10 in each. In RV clear the session. In rvshell hit the Send Image button. rvshell will create a new stereo image source in RV and send the image one tile at a time to it. The left eye will be the original image and the right eye will be its inverse. Try View→Stereo→Side by Side to see the results.

10.1.2 Using rvNetwork.py

document here

10.2 TwkQtChat Library

The TwkQtChat library is composed of three classes: Client, Connection, and Server.

`sendMessage`	Generic method to send a standard UTF-8 text message to a specific contact
`sendData`	Generic method to send a data message to a specific contact
`broadcastMessage`	Send a standard UTF-8 message to all contacts
`sendEvent`	Send an EVENT or RETURNEVENT message to a contact (calls sendMessage)
`broadcastEvent`	Send an EVENT or RETURNEVENT message to all contacts
`connectTo`	Initiate a connection to a specific contact
`hasConnection`	Query connection status to a contact
`disconnectFrom`	Force the shutdown of connection
`waitForMessage`	Block until a message is received from a specific contact
`waitForSend`	Block until a message is actually sent
`signOff`	Send a DISCONNECT message to a contact to shutdown gracefully
`online`	Returns true of the Server is running and listening on the port

Table 10.1: Important Client Member Functions

`newMessage`	A new message has been received on an existing connection
`newData`	A new data message has been received on an existing connection
`newContact`	A new contact (and associated connection) has been established
`contactLeft`	A previously established connection has been shutdown
`requestConnection`	A remote program is requesting a connection
`connectionFailed`	An attempted connection failed
`contactError`	An error occured on an existing connection

Table 10.2: Client Signals

A single Client instance is required to represent your process and to manage the Connections and Server instances. The Connection and Server classes are derived from the Qt QTcpSocket and QTcpServer classes which do the lower level work. Once the Client instance exists you can get pointer to the Server and existing Connections to directly manipulate them or connect their signals to slots in other QObject derived classes if needed.

The application should start by creating a Client instance with its contact name (usually a user name), application name, and port on which to create the server. The Client class uses standard Qt signals and slots to communicate with other code. Its not necessary to inherit from it.

The most important functions on the Client class are list in table 10.1.

10.3 The Protocol

There are two types of messages that RV can receive and send over its network socket: a standard message and a data message. Data messages can send arbitrary binary data while standard messages are used to send UTF-8 string data.

The greeting is used only once on initial contact. The standard message is used in most cases. The data message is used primarily to send binary files or blocks of pixels to/from RV.

10.3.1 Standard Messages

RV recognizes these types of standard messages:

MESSAGE	The string payload is subdivided into multiple parts the first of which indicates the sub-type of the message. The rest of the message is interpreted according to its sub-type.
GREETING	Sent by RV to a synced RV when negotiating the initial contact.
NEWGREETING	Sent by external controlling programs to RV during initial contact.
PINGPONGCONTROL	Used to negotiate whether or not RV and the connected process should exchange PING and PONG messages on a regular basis.
PING	Query the state of the other end of the connection — i.e. check and see if the other process is still alive and functioning.
PONG	Returned when a PING message is received to indicate state.

Table 10.3: Message Types

When an application first connects to RV over its TCP port, a greeting message is exchanged. This consists of an UTF-8 byte string composed of:

The string ”NEWGREETING”	1st word
The UTF-8 value 32 (space)	-
A UTF-8 integer composed of the characters [0-9] with the value $\boldsymbol{N} + \boldsymbol{M} + 1$ indicating the number of bytes remaining in the message	2nd word
The UTF-8 value 32 (space)	-
Contact name UTF-8 string (non-whitespace)	$\boldsymbol{N}$ bytes
The UTF-8 value 32 (space)	1 byte
Application name UTF-8 string (non-whitespace)	$\boldsymbol{M}$ bytes

Table 10.4: Greeting Message

In response, the application should receive a NEWGREETING message back. At this point the application will be connected to RV.

A standard message is a single UTF-8 string which has the form:

The string ”MESSAGE”	1st word
The UTF-8 value 32 (space)	-
A UTF-8 integer composed of the characters [0-9] the value of which is $\boldsymbol{N}$ indicating the size of the remaining message	2nd word
The UTF-8 value 32 (space)	-
The message payload (remaining UTF-8 string)	$\boldsymbol{N}$ bytes

Table 10.5: Standard Message

When RV receives a standard message (MESSAGE type) it will assume the payload is a UTF-8 string and try to interpret it. The first word of the string is considered the sub-message type and is used to decide how to respond:

EVENT	Send the rest of the payload as a UI event (see below)
RETURNEVENT	Same as EVENT but will result in a response RETURN message
RETURN	The message is a response to a recently received RETURNEVENT message
DISCONNECT	The connection should be disconnected

Table 10.6: Sub-Message Types

The EVENT and RETURNEVENT messages are the most common. When RV receives an EVENT or RETURNEVENT message it will translate it into a user interface event. The additional part of the string (after EVENT) is composed of:

EVENT *or* RETURNEVENT	UTF-8 string identifying the message as an EVENT or RETURNEVENT message.
space character	-
non-whitespace-event-name	The event that will be sent to the UI as a string event (e.g. `remote-eval`). This can be obtained from the event by calling `event.name()`in Mu or Python
space character	-
UTF-8 string	The string event contents. Retrievable with `event.contents()` in Mu or Python.

Table 10.7: EVENT Messages

For example the full contents of an EVENT message might look like:

```
MESSAGE 34 EVENT my-event-name red green blue
```

The first word indicates a standard message. The next word (34) indicates the lenth of the rest of the data. EVENT is the message sub-type which further specifies that the next word (my-event-name) is the event to send to the UI with the rest of the string (red green blue) as the event contents.

If a UI function that receives the event sets the return value and the message was a RETURNEVENT, then a RETURN will be sent back. A RETURN will have a single string that is the return value. An EVENT message will not result in a RETURN message.

RETURN	UTF-8 string identifying the message as an RETURN message.
space character	-
UTF-8 string	The string event `returnContents()`. This is the value set by `setReturnContents()` on the event object in Mu or Python.

Table 10.8: RETURN Message

Generally, when a RETURNEVENT is sent to your application, a RETURN should be sent back because the other side may be blocked waiting. Its ok to send an empty RETURN. Normally, RV will not send EVENT or RETURNEVENT messages to other non-RV applications. However, its possible that this could happen while connected to an RV that is also engaged in a sync session with another RV.

Finally a DISCONNECT message comes with no additional data and signals that the connection should be closed.

Ping and Pong Messages

There are three lower level messages used to keep the status of the connection up to date. This scheme relies on each side of the connection returning a PONG message if it ever receives a PING message whenever ping pong messages are active.

Whether or not its active is controlled by sending the PINGPONGCONTROL message: when received, if the payload is the UTF-8 value ”1” then PING messages should be expected and responded to. If the value is ”0” then responding to a PING message is not mandatory.

For some applications especially those that require a lot of computation (e.g. a display driver for a renderer) it can be a good to shut down the ping pong notification. When off, both sides of the connection should assume the other side is busy but not dead in the absence of network activity.

Message	**Description**	**Full message value**
PINGPONGCONTROL	A payload value of ”1” indicates that PING and PONG messages should be used	`PINGPONGCONTROL 1` (1 *or* 0)
PING	The payload is always the character ”p”. Should result in a PONG response	`PING 1 p`
PONG	The payload is always ”p”. Should be sent in response to a PING message	`PONG 1 p`

Table 10.9: PING and PONG Messages

10.3.2 Data Messages

The data messages come it two types: PIXELTILE and DATAEVENT. These take the form:

PIXELTILE(*parameters*) -*or*- DATAEVENT(*parameters*)	1st word
space character	-
A UTF-8 integer composed of the characters [0-9] the value of which is ***N*** indicating the size of the remaining message	2nd word
space character	-
Data of size ***N***	***N*** bytes

Table 10.10: PIXELTILE and DATAEVENT

The PIXELTILE message is used to send a block of pixels to or from RV. When received by RV the PIXELTILE message is translated into a `pixel-block` event (unless another event name is specified) which is sent to the user interface. This message takes a number of parameters which should have no whitespace characters and seperated by commas (”,”):

w	Width of data in pixels.
h	Height of the data in pixels.[1]
x	The horizontal offset of the pixel block relative to the image origin
y	The vertical offset of the pixel block relative to the image origin
f	The frame number
event-name	Alternate event name (instead of pixel-block). RV will only recognize pixel-block event by default. You can bind to other events however.
media	The name of the media associated with data.
layer	The name of the layer associated with the meda. This is analogous to an EXR layer
view	The name of the view associated with the media. This is analogous to an EXR view

Table 10.11: PIXELTILE Message

For example, the PIXELTILE header to the data message might appear as:

```
PIXELTILE(media=out.9.exr,layer=diffuse,view=left,w=16,h=16,x=160,y=240,f=9)
```

Which would be parsed and used to fill fields in the Event type. This data becomes available to Mu and Python functions binding to the event. By default the Event object is sent to the `insertCreatePixelBlock()` function which fins the image source associated with the meda and inserts the data into the correct layer and view of the image. Each of the keywords in the PIXELTILE header is optional.

The DATAEVENT message is similar to the PIXELTILE but is intended to be implemented by the user. The message header takes at least three parameters which are ordered (no keywords like PIXELTILE). RV will use only the first three parameters:

event-name	RV will send a raw data event with this name
target	Required but not currently used
content type string	An arbitrary string indicating the type of the content. This is available to the UI from the `Event.contentType()` function.

Table 10.12: DATAEVENT Message

For example, the DATAEVENT header might appear as:

```
DATAEVENT(my-data-event,unused,special-data)
```

Which would be sent to the user interface as a `my-data-event` with the content type ”special-data”. The content type is retrievable with `Event.contentType()`. The data payload is available via `Event.dataContents()` method.

Chapter 11

Webkit JavaScript Integration

RV can communicate with JavaScript running in a QWebView widget. This makes it possible to serve custom RV-aware web pages which can interact with a running RV. JavaScript running in the web page can execute arbitrary Mu script strings as well as receive events from RV.

You can experiment with this using the example `webview` package included with RV.

If you are not familiar with Qt's webkit integration this page can be helpful: http://doc.qt.nokia.com/4.7/qtwebkit-bridge.html.

11.1 Executing Mu or Python from JavaScript

RV exports a JavaScript object called `rvsession` to the Javascript runtime environment. Two of the functions in that namespace are `evaluate()` and `pyevaluate()`. By calling `evaluate()` or `pyevaluate()` you can execute arbitrary Mu or Python code in the running RV to control it. If the executed code returns a value, the value will be converted to a string and returned by the `(py)evaluate()` functions.

As an example, here is some html which demonstates creating a link in a web page which causes RV to start playing when pressed:

```
<script type="text/javascript">
function play () { rvsession.evaluate("play()"); }
</script>

<p><a href="javascript:play()">Play</a></p>
```

If inlining the Mu or Python code in each call back becomes onerous you can upload function definitions and even whole classes all in one evaluate call and then call the defined functions later. For complex applications this may be the most sane way to handle call back evaluation.

11.2 Getting Event Call Backs in JavaScript

RV generates events which can be converted into call backs in JavaScript. This differs slightly from how events are handled in Mu and Python.

Signal	**Events**
`eventString`	Any internal RV event and events generated by the command `sendInternalEvent()` command in Mu or Python
`eventKey`	Any `key-` event (e.g. `key-down--a`)
`eventPointer`	Any `pointer-` event (e.g. `pointer-1--push`) or tablet event (e.g. `stylus-pen--push`)
`eventDragDrop`	Any `dragdrop-` event

Table 11.1: JavaScript Signals Produced by Events

The `rvsession` object contains signal objects which you can connect by supplying a call back function. In addition you need to supply the name of one or more events as a regular expression which will be matched against incoming events. For example:

```
function callback_string (name, contents, sender)
{
    var x = name + " " + contents + " " + sender;
    rvsession.evaluate("print(\"callback_string " + x + "\\n\");");
}

rvsession.eventString.connect(callback_string);
rvsession.bindToRegex("new-source");
```

connects the function `callback_string()` to the `eventString` signal object and binds to the `new-source` RV event. For each event the proper signal object type must be used. For example pointer events are not handled by `eventString` but by the `eventPointer` signal. There are four signals available: `eventString`, `eventKey`, `eventPointer`, and `eventDragDrop`. See tables describing which events generate which signals and what the signal call back arguments should be.

In the above example, any time media is loaded into RV the `callback_string()` function will be called. Note that there is a single callback for each type of event. In particular if you want to handle both the ”new-source” and the ”frame-changed” events, your `eventString` handler must handle both (it can distinguish between them using the ”name” parameter passed to the handler. To bind the handler to both events you can call ”bindToRegex” multiple times, or specify both events in a regular expression:

```
rvsession.bindToRegex("new-source|frame-changed");
```

The format of this regular expression is specified here: http://doc.qt.nokia.com/4.7/qregexp.html

Argument	**Description**
eventName	The name of the RV event. For example ”new-source”
contents	A string containing the event contents if it has any
senderName	Name of the sender if it has one

Table 11.2: `eventString` Signal Arguments

Argument	**Description**
eventName	The name of the RV event. For example ”new-source”
key	An integer reprenting the key symbol
modifiers	An integer the low order five bits of which indicate the keyboard modifier state

Table 11.3: `eventKey` Signal Arguments

Argument	**Description**
eventName	The name of the RV event. For example ”new-source”
x	The horizonital position of the mouse as an integer
y	The veritical position of the mouse as an integer
w	The width of the event domain as an integer
h	The height of the event domain as an integer
startX	The starting horizontal position of a mouse down event
startY	The starting vertical position of a mouse down event
buttonStates	An integer the lower order five bits of which indicate the mouse button states
activationTime	The relative time at which button activation occured or 0 for regular pointer events

Table 11.4: `eventPointer` Signal Arguments

Argument	**Description**
eventName	The name of the RV event. For example ”new-source”
x	The horizonital position of the mouse as an integer
y	The veritical position of the mouse as an integer
w	The width of the event domain as an integer
h	The height of the event domain as an integer
startX	The starting horizontal position of a mouse down event
startY	The starting vertical position of a mouse down event
buttonStates	An integer the lower order five bits of which indicate the mouse button states
dragDropType	A string the value of which will be one of ”enter”, ”leave”, ”move”, or ”release”
contentType	A string the value of which will be one of ”file”, ”url”, or ”text”
stringContent	The contents of the drag and drop event as a string

Table 11.5: `eventDragDrop` Signal Arguments

11.3 Using the `webview` Example Package

This package creates one or more docked QWebView instances, configurable from the command line as described below. JavaScript code running in the webviews can execute arbitrary Mu code in RV by calling the rvsession.evaluate() function. This package is intended as an example.

These command-line options should be passed to RV after the `-flags` option. The webview options below are shown with their default values, and all of them can apply to any of four webviews in the Left, Right, Top, and Bottom dock locations.

```
shell> rv -flags ModeManagerPreload=webview
```

The above forces the load of the webview package which will display an example web page. Additional arguments can be supplied to load specific web pages into additional panes. While this will just show the sample html/javascript file that comes with the package in a webview docked on the right. To see what's happening in this example, bring up the Session Manager so you can see the Sources appearing and disappearing, or switch to the defaultLayout view. Note that you can play while reconfiguring the session with the javascript checkboxes.

The following additional arguments can be passed via the `-flags` mechanism. In the below, ***POS*** should be replaced by one of *Left*, *Right*, *Bottom*, or *Top*.

ModeManagerPreload=webview Force loading of the webview package. The package should not be loaded by default but does need to be installed. This causes rv to treat the package as if it were loaded by the user.

webviewUrl*POS*=*URL* A webview pane will be created at POS and the URL will be loaded into it.

webviewTitle*POS*=*string* Set the title of the webview pane to string.

webviewShowTitle*POS*=*true* or *false* A value of true will show and false will remove the title bar from the webview pane.

webviewShowProgress*POS*=*true* or *false* Show a progress bar while loading for the web pane.

webviewSize*POS*=*integer* Set the width (for right and left panes) or height (for top and bottom panes) of the web pane.

An example using all of the above:

```
shell> rv -flags ModeManagerPreload=webview \
      webviewUrlRight=file:///foo.html \
      webviewShowTitleRight=false \
      webviewShowProgressRight=false \
      webviewSizeRight=200 \
      webviewUrlBottom=file:///bar.html \
      webviewShowTitleBottom=false \
      webviewShowProgressBottom=false \
      webviewSizeBottom=300
```

Chapter 12

Hierarchical Preferences

Each RV user has a Preferences file where her personal rv settings are stored. Most preferences are viewed and edited with the Preferences dialog (accessed via the RV menu), but preferences can also be programmatically read and written from custom code via the `readSetting` and `writeSetting` Mu commands. The preferences files a stored in different places on different platforms.

Platform	Location
Mac OS X	\$HOME/Library/Preferences/com.tweaksoftware.RV.plist
Linux	\$HOME/.config/TweakSoftware/RV.conf
Windows XP	\$HOME/Application Data/TweakSoftware/RV.ini
WIindows Vista/7	˜\$HOME/AppData/Roaming/TweakSoftware/RV.ini

Initial values of preferences can be overridden on a site-wide or show-wide basis by setting the environment variable RV_PREFS_OVERRIDE_PATH to point to one or more paths that contain files of the name and type listed in the above table. Each of these overriding preferences file can provide default values for one or more preferences. A value from one of these overriding files will override the users's preference only if the user's preferences file has no value for this preference yet.

In the simplest case, if you want to provide overriding initial values for all preferences, you should

1. Delete your preferences file.
2. Start RV, go to the Preferences dialog, and adjust any preferences you want.
3. Close the dialog and exit RV.
4. Copy your preferences file into the RV_PREFS_OVERRIDE_PATH.

If you want to override at several levels (say per-site and per-show), you can add preferences files to any number of directories in the PATH, but you'll have to edit them so that each only contains the preferences you want to override with that file. Preferences files found in directories earlier in the path will override those found in later directories.

Note that this system only provides the ability to override initial settings for the preferences. Nothing prevents the user from changing those settings after initialization.

It's also possible to create show/site/whatever-specific preferences files that **always** clobber the user's personal preferences. This mechanism is exactly analogous to the above, except that the name of the environment variable that holds paths to clobbering prefs files is RV_PREFS_CLOBBER_PATH. Again, the user can freely change any "live" values managed in the Preferences dialog, but in the next run, the clobbering preferences will again take precedence. Note that a value from a clobbering file (at any level) will take precedence over a value from an overriding file (at any level).

Chapter 13

Node Reference

This chapter has a section for each type of node in RV's image processing graph. The properties and descriptions listed here are the default properties.

RVSourceGroup

The source group contains a single chain of nodes the leaf of which is an RVFileSource or RVImageSource. It has a single property.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.

RVSequenceGroup

The sequence group contains a chain of nodes for each of its inputs. The input chains are connected to a single RVSequence node which controls timing and switching between the inputs.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.
timing.retimeInputs	int	1	Retime all inputs to the output fps if 1 otherwise play back their frames one at a time at the output fps.

RVStackGroup

The stack group contains a chain of nodes for each of its inputs. The input chains are connected to a single RVStack node which controls compositing of the inputs as well as basic timing offsets.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.
timing.retimeInputs	int	1	Retime all inputs to the output fps if 1 otherwise play back their frames one at a time at the output fps.

RVSwitchGroup

The switch group changes it behavior depending on which of its inputs is ”active”. It contains a single Switch node to which all of its inputs are connected.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.

RVRetimeGroup

The source group contains a single chain of nodes the leaf of which is an RVFileSource or RVImageSource. It has a single property.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.

RVLayoutGroup

The source group contains a single chain of nodes the leaf of which is an RVFileSource or RVImageSource. It has a single property.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.
timing.retimeInputs	int	1	Retime all inputs to the output fps if 1 otherwise play back their frames one at a time at the output fps.

RVFolderGroup

The folder group contains either a SwitchGroup or LayoutGroup which determines how it is displayed.

Name	Type	Size	Description
ui.name	string	1	This is a user specified name which appears in the user interface.
mode.viewType	string	1	Either ”switch” or ”layout”. Determines how the folder is displayed.

RVAdapter

This node has no properties.

RVFileSource

The source node controls file I/O and organize the source media into layers (in the RV sense). It has basic controls needed to mix the layers together.

Name	Type	Size	Description
media.movie	string	> 1	The movie, image, audio files and image sequence names. Each name is a layer in the source.There is typically at least one value in this property
group.fps	float	1	Overrides the fps found in any movie or image file or if none is found overrides the default fps of 24.
group.volume	float	1	Relative volume. This can be any positive number or 0.
group.audioOffset	float	1	Audio offset in seconds. All audio layers will be offset.
group.rangeOffset	int	1	Shifts the start and end frame numbers of all image media in the source.
group.balance	float	1	Range of [-1,1]. A value of 0 means the audio volume is the same for both the left and right channels.
group.crossover	float	1	Range of [0, 1]. 0 means no cross over, 1 means swap the channels.
group.noMovieAudio	int	1	Do not use audio tracks in movies files
cut.in	int	1	The preferred start frame of the sequence/movie file
cut.out	int	1	The preferred end frame of the sequence/movie file
request.readAllChannels	int	1	If the value is 1 and the image format can read multiple channels, it is requested to read all channels in the current image layer and view.
request.imageLayerSelection	string	Any	Any values are considered image layer names. These are passed to the image readers with the request that only these layers be read from the file.
request.imageViewSelection	string	Any	Any values are considered image view names. These are passed to the image readers with the request that only these views be read from the file.
request.stereoViews	string	0 or 2	If there are values in this property, they will be passed to the image reader when in stereo viewing mode as requested view names for the left and right eyes.

RVFormat

Property	Type	Size	Description
geometry.xfit	int	1	Used by RVIO. Forces the resolution to a specific width
geometry.yfit	int	1	Used by RVIO. Forces the resolution to a specific height
geometry.scale	float	1	Multiplier on incoming resolution. E.g., 0.5 when applied to 2048x1556 results in a 1024x768 image.
geometry.resampleMethod	string	1	Method to use when resampling. The possible values are area, cubic, and linear,
crop.active	int	1	If non-0 cropping is active
crop.xmin	int	1	Minimum X value of crop in pixel space
crop.ymin	int	1	Minimum Y value of crop in pixel space
crop.xmax	int	1	Maximum X value of crop in pixel space
crop.ymax	int	1	Maximum Y value of crop in pixel space
uncrop.active	int	1	In non-0 uncrop region is used
uncrop.x	int	1	X offset of input image into uncropped image space
uncrop.y	int	1	Y offset of input image into uncropped image space
uncrop.width	int	1	Width of uncropped image space
uncrop.height	int	1	Height of uncropped image space
color.maxBitDepth	int	1	One of 8, 16, or 32 indicating the maximum allowed bit depth (for either float or integer pixels)
color.allowFloatingPoint	int	1	If non-0 floating point images will be allowed on the GPU otherwise, the image will be converted to integer of the same bit depth (or the maximum bit depth).

RVChannelMap

Property	Type	Size	Description
format.channels	string	>= 0	An array of channel names. If the property is empty the image will pass though the node unchanged. Otherwise, only those channels appearing in the property array will be output. The channel order will be the same as the order in the property.

RVHistogram

The histogram node computes a channel histogram and stores each one as a separate image attribute

Property	Type	Size	Description
node.active	int	1	non-0 means the node is active and will compute a histogram
histogram.size	int	1	Number of histogram buckets per channel. Defaults to 100.

RVCache

The RVCache node has no external properties.

RVTransform2D

The 2D transform node controls the image transformations. This node is usually evaluated on the GPU.

Property	Type	Size	Description
transform.flip	int	1	non-0 means flip the image (vertically)
transform.flop	int	1	non-0 means flop the image (horizontally)
transform.rotate	float	1	Rotate the image in degrees about its center.
pixel.aspectRatio	float	1	If non-0 set the pixel aspect ratio. Otherwise use the pixel aspect ratio reported by the incoming image.
transform.translate	float[2]	1	Translation in 2D in NDC space
transform.scale	float[2]	1	Scale in X and Y dimensions in NDC space
stencil.visibleBox	float	4	Four floats indicating the left, right, top, and bottom in NDC space of a stencil box.

RVColor

The color node has a large number of color controls. This node is usually evaluated on the GPU, except when normalize is 1.

Property	Type	Size	Description
color.alphaType	int	1	By default (0), uses the alpha type reported by the incoming image. Otherwise, 1 means the alpha is premultiplied, 0 means the incoming alpha is unpremultiplied.
color.normalize	int	1	Non-0 means to normalize the incoming pixels to [0,1]
color.logtype	int	1	The default (0), means no log to linear transform, 1 uses the cineon transform (see cineon.whiteCodeValue and cineon.blackCodeValue below) and 1 means use the Viper camera log to linear transform.
color.invert	int	1	If non-0, invert the image color using the inversion matrix (See User's Manual)
color.sRGB2linear	int	1	If the value is non-0, convert the incoming pixels from sRGB space to linear space
color.Rec709ToLinear	int	1	If the value is non-0, convert the incoming pixels using the inverse of the Rec709 transfer function
color.gamma	float[3]	1	Apply a gamma to the incoming image. The default is [1.0, 1.0, 1.0]. The three values are applied to R G and B channels independently.
color.lut			Not currently used
color.offset	float[3]	1	Color bias added to incoming color channels. Default = 0 (not bias). Each component is applied to R G B independently.
color.exposure	float[3]	1	Relative exposure in stops. Default = [0, 0, 0], See user's manual for more information on this. Each component is applied to R G and B independently.
color.contrast	float[3]	1	Contrast applied per channel (see User's Manual)
color.saturation	float	1	Relative saturation (see User's Manual)
color.hue	float	1	Hue rotation in radians (see User's Manual)
color.active	int	1	If 0, do not apply any color transforms. Disables the node.
color.ignoreChromaticities	int	1	If non-0, ignore any non-Rec 709 chromaticities reported by the incoming image.
CDL.slope	float[3]	1	Color Decision List per-channel slope control
CDL.offset	float[3]	1	Color Decision List per-channel offset control
CDL.power	float[3]	1	Color Decision List per-channel power control
CDL.saturation	float[3]	1	Color Decision List saturation control
luminanceLUT.lut	float	div 3	Luminance LUT to be applied to incoming image. Contains R G B triples one after another. The LUT resolution
luminanceLUT.max	float	1	A scale on the output of the Luminance LUT
luminanceLUT.size	int	1	The size of the luminance LUT
luminanceLUT.type	string	1	Only acceptable value is currently "Luminance"
luminanceLUT.active	int	1	If non-0, luminance LUT should be applied
cineon.whiteCodeValue	int	1	Value used in cineon log to linear conversion. Default is 685
cineon.blackCodeValue	int	1	Value used in cineon log to linear conversion. Default is 95
luminanceLUT:output.size	int	1	Output Luminance lut size
luminanceLUT:output.lut	float	div 3	Output resampled luminance LUT
matrix:output.RGBA	float	16	Output color matrix
lut.lut	float	div 3	Contains either a 3D or a channel file LUT
lut.prelut	float	div 3	Contains a channel pre-LUT
lut.inMatrix	float	16	Input color matrix
lut.scale	float	1	LUT output scale factor
lut.offset	float	1	LUT output offset
lut.max	float	1	A scaling value on the output of the channel LUT
lut.name	string	1	Placeholder used for annotation
lut.file	string	1	Name of LUT file to read when RV session is loaded
lut.size	int	1 or 3	With 1 size value, the file LUT is a channel LUT of the specified size, if there are 3 values the file LUT is a 3D LUT with the dimensions indicated
lut.active	int	1	If non-0 the file LUT is active
lut:output.size	int	1 or 3	The resampled LUT output size
lut:output.lut	float	div 3	The resampled output LUT

RVLookLUT and RVCacheLUT

The RVCacheLUT is applied in software before the image is cached and before any software resolution and bit depth changes. The look LUT is applied just before the display LUT but is per-source.

Property	Type	Size	Description
lut.lut	float	div 3	Contains either a 3D or a channel look LUT
lut.prelut	float	div 3	Contains a channel pre-LUT
lut.inMatrix	float	16	Input color matrix
lut.scale	float	1	LUT output scale factor
lut.offset	float	1	LUT output offset
lut.max	float	1	A scaling value on the output of the channel LUT
lut.name	string	1	Placeholder used for annotation
lut.file	string	1	Name of LUT file to read when RV session is loaded
lut.size	int	1 or 3	With 1 size value, the look LUT is a channel LUT of the specified size, if there are 3 values the look LUT is a 3D LUT with the dimensions indicated
lut.active	int	1	If non-0 the look LUT is active
lut:output.size	int	1 or 3	The resampled LUT output size
lut:output.lut	float	div 3	The resampled output LUT

RVSequence

Information about how to create a working EDL can be found in the User's Manual. All of the properties in the edl component should be the same size.

Property	Type	Size	Description
edl.frame	int	N	The global frame number which starts each cut
edl.source	int	N	The source input number of each cut
edl.in	int	N	The source relative in frame for each cut
edl.out	int	N	The source relative out frame for each cut
edl.inc	int	N	Frame increment relative to the local source frame **DEPRECATED**
edl.action	string	N	Currently, "play" should be the used here in all cases, except the last edit which should be "end". **DEPRECATED**
output.fps	float	1	Output FPS for the sequence. Input nodes may be retimed to this FPS.
output.size	int[2]	1	The virtual output size of the sequence. This may not match the input sizes.
output.interactiveSize	int	1	If 1 then adjust the virtual output size automatically to the window size for framing.
output.autoSize	int	1	Figure out a good size automatically from the input sizes if 1. Otherwise use output.size.
mode.useCutInfo	int	1	Use cut information on the inputs to determine EDL timing.
mode.autoEDL	int	1	If non-0, automatically concatenate new sources to the existing EDL, otherwise do not modify the EDL

RVStack

Property	Type	Size	Description
output.fps	float	1	Output FPS for the stack. Input nodes may be retimed to this FPS.
output.size	int[2]	1	The virtual output size of the stack. This may not match the input sizes.
output.autosize	int	1	Figure out a good size automatically from the input sizes if 1. Otherwise use output.size.
output.chosenAudioInput	string	1	Name of input which becomes the audio output of the stack. If the value is `.all.` then all inputs are mixed. If the value is `.first.` then the first input is used.
composite.type	string	1	The compositing operation to perform on the inputs. Valid values are: `over`, `add`, `difference`, `-difference`, and `replace`
mode.useCutInfo	int	1	Use cut information on the inputs to determine EDL timing.
mode.alignStartFrames	int	1	If 1 offset all inputs so they start at same frame as the first input.

RVSwitch

Property	Type	Size	Description
output.fps	float	1	Output FPS for the switch. This is normally determined by the active input.
output.size	int[2]	1	The virtual output size of the stack. This is normally determined by the active input.
output.autosize	int	1	Figure out a good size automatically from the input sizes if 1. Otherwise use output.size.
output.input	string	1	Name of the active input node.
mode.useCutInfo	int	1	Use cut information on the inputs to determine EDL timing.
mode.alignStartFrames	int	1	If 1 offset all inputs so they start at same frame as the first input.

RVSoundTrack

Property	Type	Size	Description
audio.volume	float	1	Global audio volume
audio.balance	float	1	[-1,1] left/right channel balance
audio.offset	float	1	Globl audio offset in seconds
audio.mute	int	1	If non-0 audio is muted

RVStereo

Property	Type	Size	Description
stereo.swap	int	1	If non-0 swap the left and right eyes
stereo.relativeOffset	float	1	Offset distance between eyes, default = 0. Both eyes are offset.
stereo.rightOffset	float	1	Offset distance between eyes, default = 0. Only right eye is offset.
rightTransform.flip	int	1	If non-0 flip the right eye
rightTransform.flop	int	1	If non-0 flop the right eye
rightTransform.rotate	float	1	Right eye rotation in radians
rightTransform.translate	float[2]	1	independent 2D translation applied only to right eye (on top of offsets)

RVDisplayStereo

Property	Type	Size	Description
stereo.type	string	1	

RVRetime

Property	Type	Size	Description
visual.scale	float	1	
visual.offset	float	1	
audio.scale	float	1	
audio.offset	float	1	
output.fps	float	1	

RVComposite

This node has been deprecated in version 3.10. The RVStack node subsumes it.

RVDispTransform2D

Property	Type	Size	Description
transform.translate	float[2]	1	Viewing translation
transform.scale	float[2]	1	Viewing scale

RVDisplay

Property	Type	Size	Description
color.channelOrder	string	1	A four character string containing any of the characters [RGBA10]. The order allows permutation of the normal R G B and A channels as well as filling any channel with 1 or 0.
color.channelFlood	int	1	If 0 pass the channels through as they are. When the value is 1, 2, 3, or 4, the R G B or A channels are used to flood the R G and B channels. When the value is 5, the luminance of each pixel is computed and displayed as a gray scale image.
color.gamma	float	1	A single gamma value applied to all channels, default = 1.0
color.sRGB	int	1	If non-0 a linear to sRGB space transform occurs
color.Rec709	int	1	If non-0 the Rec709 transfer function is applied
color.brightness	float	1	In relative stops, the final pixel values are brightened or dimmed according to this value. Occurs after all color space transforms.
color.outOfRange	int	1	If non-0 pass pixels through an out of range filter. Channel values in the (0,1] are set to 0.5, channel values [-inf,0] are set to 0 and channel values (1,inf] are set to 1.0.
color.active	int	1	If 0 deactivate the display node
lut.lut	float	div 3	Contains either a 3D or a channel display LUT
lut.prelut	float	div 3	Contains a channel pre-LUT
lut.scale	float	1	LUT output scale factor
lut.offset	float	1	LUT output offset
lut.inMatrix	float	16	Input color matrix
lut.max	float	1	A scaling value on the output of the channel LUT
lut.name	string	1	Placeholder used for annotation
lut.file	string	1	Name of LUT file to read when RV session is loaded
lut.size	int	1 or 3	With 1 size value, the display LUT is a channel LUT of the specified size, if there are 3 values the display LUT is a 3D LUT with the dimensions indicated
lut.active	int	1	If non-0 the display LUT is active
lut:output.size	int	1 or 3	The resampled LUT output size
lut:output.lut	float	div 3	The resampled output LUT