FreeJ User's Guide

freej.dyne.org

Jaromil
RASTASOFT

FreeJ User's Guide: freej.dyne.org

by Jaromil

Table of Contents

Chapter 1. Introduction

FreeJ is a vision mixer: an instrument for realtime video manipulation used in the fields of dance teather, veejaying, medical visualisation and TV.

With FreeJ multiple layers can be filtered thru effect chains and then mixed together. The supported layer inputs are images, movies, live cameras, particle generators, text scrollers and more. All the resulting video mix can be shown on multiple and remote screens, encoded into a movie and streamed live to the internet.

FreeJ can be controlled locally or remotely, also from multiple places at the same time, using its slick console interface; can be automated via javascript and operated via MIDI and Joystick. Internally, FreeJ handles video data in RGBA colorspace, that is: 32bit wide collection of 4 channels, 8bit each, representing the red, green, blue and alpha components.

This manual will guide you to the usage of this application, an up to date version of it can be found on the FreeJ website[1].

This is Free software

Nobody should be restricted by the software they use. There are four freedoms that every user should have:

the freedom to use the software for any purpose,
the freedom to change the software to suit your needs,
the freedom to share the software with your friends and neighbors, and
the freedom to share the changes you make.

When a program offers users all of these freedoms, we call it free software.

1. http://freej.dyne.org

Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

This is Rasta software

Jah Rastafari Livity bless our Freedom! This is free software, share it for the good of yourself and your people, respect others and let them express, be free and let others be free. Live long and prosper in Peace!

But, no Peace without Justice. This software is about Resistance inna babylon world which tries to control more and more the way we communicate and we share informations and knowledge. This software is for all those who cannot afford to have the latest expensive hardware to speak out their words of consciousness and good will. This software has a full range of applications for production and not only fruition of information, it's a full multimedia studio, you don't need to buy anything to express your voice. Freedom and sharing of knowledge are solid principles for evolution and that's where this software comes from.

Inna babylon, money is the main requirement to make a voice possible to be heard by others. Capitalist and fundamentalist governments all around the world rule with huge TV monopolies spreading their propaganda, silencing all criticism.

This is a struggle for Redemption from existing operating systems which always require new expensive hardware for doing the same as ever: give us free players but make us pay for producing our own voices. And the one who protects you rips you off, as the Arabs say.

FreeJ is a tool to produce and publish yourself, freely. There is nothing to consume here, there is all you need to create.

Commercial operating systems always give a possibility to listen - all kinds of "free to download" players, but always with restrictions and no easy way for everybody to speak out. The way communication is structured follows the hierarchy of powers allready established in babylon's mediascapes and, worst than ever, money is the main requirement to spread a voice and let it be heard by others.

Neverthless, proprietary software spreads the dependence from business companies thru the populace: whenever we share our knowledge on how to use a certain software, we make the people in need to buy the tools from merchants in order to express their creativity. This is great responsability for anyone of us who teaches somebody how to do something with software: the need to buy will be slavery under the merchantile interests of capitalism.

The roots of Rasta culture can be found in Resistance to slavery. This software is not a business. This software is free as of speech and is one step in the struggle for Redemption and Freedom. This software is dedicated to the memory of Patrice Lumumba, Marcus Garvey, Marthin Luther King, Steve Biko, Walter Rodney, Malcom X; in solidarity with Mumia Abu Jamal and all those who still resist to slavery, racism and oppression, who still fight imperialism and seek an alternative to the hegemony of capitalism in our World.

Hic Sunt Leones[2]. And Much Blessings in Jah Luv to All Those who still Resist. Selah.

Disclaimer

Thanks for reviewing and inspirations to this documentation go to people that collaborated documenting FreeJ during all these years: Anne-Marie Skriver, Marloes de Valk, Robert de Geuss, Piotr Sobolewski, Alejo Duque and more...

2. http://rastasoft.org

being invariant, with the Front-Cover and Back-Cover Texts clearly stating authorship. You should have received a copy of the GNU Free Documentation License along with this manual; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

FreeJ is copyleft (GNU GPLv3)

Chapter 2. Install

Distributed packages

FreeJ packages are available on mainstream GNU/Linux distributions as Debian, Ubuntu and Fedora, successfully compiled for x86, 64bit, MIPS, PPC and even more hardware platforms: to install them you can use your favorite software manager like Synaptic or apt-get, something like "*apt-get install freej*" will work.

FreeJ is made out of multiple components, basically consisting of a library, one or more application interfaces and one or more language bindings. At the time of this publication the following packages are commonly found:

- *freej* main command-line interface (CLI)
- *freej-dbg* CLI debugging symbols
- *freej-doc* documentation and examples
- *libfreej0* run-time library
- *libfreej-dev* development headers
- *libfreej0-dbg* library debugging symbols
- *python-freej* python language bindings
- *python-freej-dbg* python debugging symbols

Compile from source

Stable sources for FreeJ are regularly published on ftp.dyne.org[1] server and its mirrors.

```
ftp://ftp.dyne.org/freej/releases
```

You can use stable releases in case you want to employ this software in a production environment, although some newest features and fixes will be missing.

If you are subscribed to the FreeJ mailinglist[2] and keen to share help with the community, then please install and run your tests on a source code fresh from the git.dyne.org[3] repository, to be able to interact with people active on making this software better. To download the latest FreeJ source then install the code revisioning system git[4] in your system, then run from a terminal:

```
git clone git://code.dyne.org/freej.git
```

This will download all the revisioned source repository on your harddisk, with an occupation lower than 50MB, which you can later update running the command 'git pull --rebase' from inside the directory created.

In case you are invited to join development and submit your modifications to the code hosted on dyne.org[5], then be introduced to our way to interact by the Developer's Lounge[6].

1. ftp://ftp.dyne.org/freej/releases
2. http://lists.dyne.org
3. http://git.dyne.org
4. http://git.or.cz
5. http://dyne.org
6. http://lab.dyne.org/Code

Dependencies

To compile FreeJ source code you will need to fulfill a minimum set of dependencies, while optionally you can link a range of libraries for extended functionality as video playback, encoding and streaming.

The Minimum requirements are:

A sane GNU C++ compiler toolchain
SDL libraries for visualization

Having just those the source code can be tweaked to compile into a minimal version of FreeJ, still missing many features.

The recommended components are:

- *FFmpeg* libraries for video playback
- *Ogg/Theora/Vorbis* libraries for video encoding and streaming
- *XUL-runner* libraries for a recent javascript interpreter
- *Jack and FFTW3* for audio analysis and parametrization
- *Swig* code wrapper for language bindings
- *Python* (and ipython console) for the language bindings
- *OpenGL* libraries for video output in 3d context

In case your building system is a GNU/Linux distribution, then make sure you have installed the corresponding -dev packages.

Build on GNU/Linux

Once having attained the source code of FreeJ we can proceed to its compilation, but first make sure all necessary development packages area available, on Debian and Ubuntu operating systems those are:

- *pkg-config* manage compile and link flags for libraries
- *flex* a fast lexical analyzer generator.
- *bison* a parser generator that is compatible with YACC
- *libsdl-dev* Simple Direct Media layer package
- *libpng-dev* Portable Network Graphics package
- *libjpeg-dev* Jpeg image development package
- *libfreetype6-dev* FreeType 2 font engine
- *libfontconfig-dev* font configuration package
- *libogg-dev* Ogg Bitstream Library Development
- *libvorbis-dev* The Vorbis General Audio Compression Codec
- *libtheora-dev* The Theora Video Compression Codec
- *libslang2-dev* The S-Lang programming library
- *libavutil-dev* FFMpeg development utilities
- *libavcodec-dev* FFMpeg development codecs

- *libavformat-dev* FFMpeg development format handling
- *libswscale-dev* FFMpeg development filter handling
- *libunicap2-dev* unified interface to video capture devices
- *libbluetooth-dev* BlueZ Linux Bluetooth library
- *fftw3-dev* Fast Fourier Transform library
- *libjack-dev* JACK Audio Connection Kit
- *libasound-dev* Alsa sound libraries
- *libmozjs-dev* Mozilla SpiderMonkey JavaScript library
- *xulrunner-dev* Gecko engine library
- *python-all-dev* Python development packages
- *python-central* register and build utility for Python packages
- *swig* Generate scripting interfaces to C/C++ code
- *libhtml-template-perl* HTML Templates with Perl for generated documentation

After having uncompressed the FreeJ source code into its directory, go inside and type:

```
./configure --enable-python
make
sudo make install
```

If no errors occurred then you should have at this point all FreeJ libraries and language bindings compiled and installed in */usr/local* directories. To test the installation type *freej* in a console and see if it starts, or run a python interpreter as *ipython* and give it the command *import freej;*.

Build on Apple/OSX

If you are trying to compile this software on OSX, you will need to open an account as "Apple Developer" and download the XCode developers tools. These are not really free, their use is a concession by the Apple corporation even if they include the GNU C compiler. Also you will need to install the free packaged collection of software Macports[7] and proceed installing the needed packages: .

```
sudo port install slang2 fftw-3 spidermonkey
sudo port install libsdl libsdl_image libsdl_ttf libsdl_gfx
sudo port install ffmpeg-devel +no_nonfree
```

If you think you are set, you can open the XCode project for FreeJ inside the osx/ directory, where there are implemented native Carbon/Cocoa OSX components used internally by our engine, with well satisfying results.

You don't need to do all this to run FreeJ on Apple/OSX: the user-friendly packaging of stable binary releases for OSX is hosted on our web page at the address freej.dyne.org[8]

7. http://www.macports.org
8. http://freej.dyne.org

Build on Microsoft/Win

No-one of us so far felt like compiling FreeJ on the M$ platform, while there is a quick and easy way to try this software using the dyne:bolic liveCD[9] even without installing anything. If you are an hacker and you know what you are doing please take contact with developers on the FreeJ mailinglist[10], we'll be interested to include patches: some minimum porting adjustment needs to be done, but basically the code is multi-platform (POSIX.1b) and will run; development packages for the dependencies should be provided already by the MinGW[11] project.

Build for embedded platforms

Experimental builds of FreeJ have succeeded to work on common game consoles. If you are trying to compile on embedded ARM, you'll probably have some fun :) the autoconf/automake setup will work out of the box with a sane cross-compiling toolchain based on GCC[12], like the one provided by devkitPro[13]: if properly linked to the right libraries and with some minor adjustement to code this software can run on game consoles like NDS or GP2X, set-top TV boxes and some palmar devices as personal managers and new generation phones. Let's get in touch: we are interested in further development in this direction!

9. http://dynebolic.org
10. http://lists.dyne.org
11. http://www.mingw.org
12. http://gcc.gnu.org
13. http://www.devkitpro.org

Chapter 3. Scripting in Javascript

Asynchronous and Object Oriented

FreeJ is an asynchronous video rendering engine that can be scripted using javascript syntax (ECMA script) in an object oriented way, to control its operations thru a procedural list of commands and actions. This way we started talking in 2005 about "procedural video" as an evolution of the current non/linear paradigm widely spread in video production.

Far from starting with such an high theoretical approach, this document aims to be a small and effective introduction to the first free software implementation of such a powerful functionality. It will not cover the programming syntax, but simply describe the objects that are made available to script FreeJ. Knowledge of generic object oriented programming syntax is suggested but not strictly necessary to make the first steps in the wonderful world of Procedural Video Scripting :)

FreeJ scripting is completely object oriented, meaning that almost every operation is provided by objects that can be created: there are no global functions, but commands that are strictly related to a certain object.

Layers, Effects and VideoEncoders are the object classes we can deal with. Once they exist, they will provide methods to control their operations. This feature represents the most stable way to script FreeJ operations, based on an implementation of the SpiderMonkey parser 1.6.

Every Layer inherits generic methods that can be commonly found, as for instance: set/get_position, set/get_blit, zoom, rotate etc.. while specific methods are found on particular layers, as they can control special operations like: print to a word on the text layer, pause for the movie layer and so on.

A fairly complete reference for FreeJ scripting is found on the freej.dyne.org online documentation[1], always check that for a detailed description of the API and get in touch with the mailinglist if you have suggestions.

Hello world

Hereby provided the hello world example: 3 lines in our case. The TextLayer is provided in FreeJ when compiled with the FreeType libraries.

```
text = new TxtLayer();
```

1. http://freej.dyne.org/docs/scripting

we create a new TextLayer[2]

```
text.print("Hello world!");
```

will print the Hello world! message in the text layer

```
add_layer(text);
```

will make the new layer shown on the screen

You can try this 3 lines script by writing it into a file and then executing it with 'freej -j helloworld' or even giving live commands inside the freej console, pressing ctrl-x and then typing in every line.

2. http://freej.dyne.org/docs/scripting/TextLayer.html

Chapter 4. Stream to the masses

FreeJ can produce audio/video streams and broadcast them to the internet, as well save the encoded streams into local files. Following a recursive scheme of re-transmission, FreeJ can also play internet streams and re-mix them and re-stream...

Since the 0.10 release of FreeJ the way to do this is by configuring stream parameters in a javascript and then loading it from a running FreeJ instance (pressing ctrl-j in the console) or from another script (starting the stream script with freej -j).

A typical streaming configuration looks like this:

```
// create a video encoder object
//    values 1-100         video quality  video bitrate  audio quality  audio_bitrate
encoder = new VideoEncoder(10,             64000,        5,             24000);
encoder.stream_host("giss.tv");
encoder.stream_port(8000);
encoder.stream_title("testing new freej");
encoder.stream_username("source");
encoder.stream_password("2t645");
encoder.stream_mountpoint("freej-test.ogg");

register_encoder(encoder);
encoder.start_stream();
// encoder.start_filesave("prova.ogg");
```

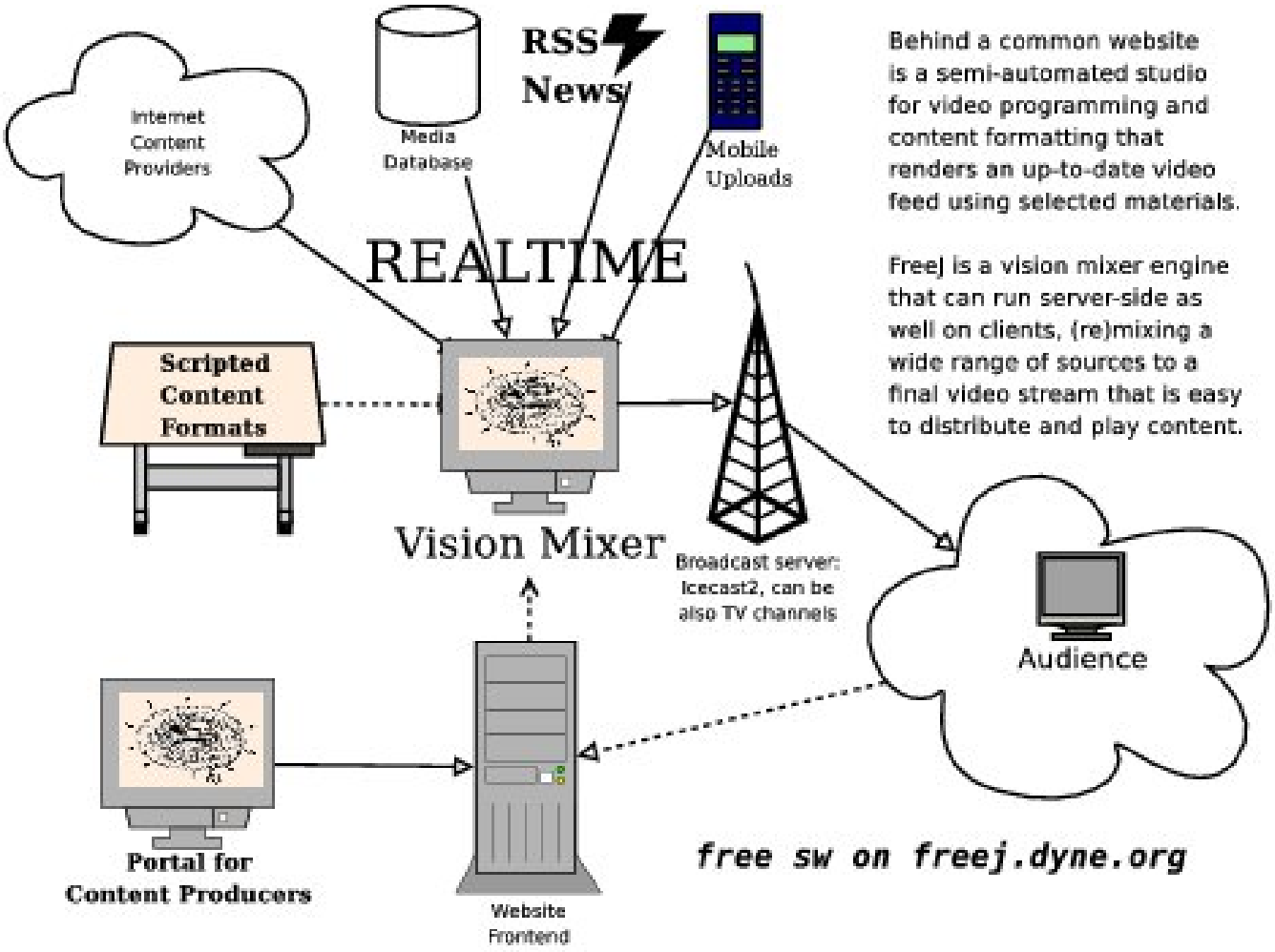
Up to date information on how to include FreeJ into a webpage is provided on the online documentation wiki[1], where you'll also find links to free broadcasting servers and clients to playback.

Remote Vision Mixer

FreeJ can be easily scripted to execute various operations on a remote computer, like mixing together video streams, rendering RSS news feeds and logos on them and re-stream the result to the public.

The advantages of running such a setup server-side are several: multiple people can manage the system remotely, while the engine will keep running an online stream, here below is a scheme:

1. http://lab.dyne.org/FreejStreaming



Server-side setup diagram for FreeJ

Chapter 5. Use in Python

You can use FreeJ in your python code to playback videos quite efficiently: it's API has been designed in C++ and the namespace tends to be clear and intuitive. Using Python bindings for the FreeJ engine can give good results, also when used in conjunction with frameworks as PyGame and Crystal Space.

To get started, first make sure you have installed a version of FreeJ compiled with python bindings: that means a package like *python-freej* in Debian or Ubuntu (see the *Install* section of this manual), or a fresh source compiled with Swig and Python development packages. FreeJ should have installed freej.py in /usr/lib (or /usr/local/lib depending from your installation prefix) inside python/site-packages.

Hello World

Here below the classical "Hello World" script that simply opens up a FreeJ output window and a Text Layer, writing into it,

```
# system wide useful modules
import threading
import time

import freej

# initializes FreeJ creating a Contex
cx = freej.Context()

# creates a screen of given size
scr = freej.SdlScreen( 400, 300 )

# adds the screen
cx.add_screen(scr)

# create an instance of a TextLayer
txt = freej.TextLayer()

# initializes the new layer with the freej context
txt.init(cx);

# writes the hello world text inside the layer
txt.write("Hello World!")
```

```
# start the layer
txt.start();

# add the layer to the screen
cx.add_layer(txt);

# starts freej in a separate thread
th = threading.Thread(target = cx.start , name = "freej")
th.start();
```

After initialisation of *cx* as a FreeJ context a *TextLayer* is initialised and added to it. Please note the sequence for creating the layer is fixed: first the constructor, then *init()* is called passing the context as an argument, some text is written into it using the *write()* method, the layer is then started and added to the context.

Play and apply an effect

The following script should be run with an argument: an image or video file that will be reproduced with a filter effect applied on it.

```
  # system wide useful modules
import threading
import time
import sys

import freej

  # initializes FreeJ creating a Contex
cx = freej.Context()

  # creates a screen of given size
scr = freej.SdlScreen( 400, 300 )

        # adds the screen
cx.add_screen(scr)

  # refreshes the list of available filter effects
cx.plugger.refresh(cx)

  # check that we have an argument
if(sys.argv.__len__()<2):
    print "[!] this script needs an argument: file to play"
    quit()

  # opens the file given on commandline as a layer
lay = cx.open(sys.argv[1])


  # gets the vertigo filter effect
filt = cx.filters["vertigo"]

  # adds the filter to the layer
lay.add_filter( filt )

  # start the layer thread
lay.start()
```

```
  # adds the layer to the freej context
cx.add_layer(lay)

  # starts freej in a separate thread
th = threading.Thread(target = cx.start , name = "freej")
th.start();
```

This script creates and initialises a context, that is an instance of FreeJ, with a size of 400 by 300; the size can also be adjusted interactively later, with screen implementations that support it. It then completes initialisation of plugin effects refreshing the list of those present on the system using *plugger.refresh*.

It then opens the file given at commandline as first argument (argv) and starts it in a layer, adds the visual effect "vertigo" to the layer created, adds the layer to the screen and runs.

However, the above example is not all you can already do with FreeJ: all the C++ API is mapped to various languages and you can use it as it is done in its code, which right now is the main documentation source, already reasonably documented via doxygen. For example, studying the header of the video_encoder implementation and how it is used in the C++ code, you can recognize the classes that are accessible from Python and script the encoding and streaming capabilities. There are many possibilities out there ;^)

Controllers

Controllers in FreeJ are listeners which will dispatch asynchronous messages from input channels, that is for example a keyboard that will activate callbacks every time keys are pressed, informing the running script that something is happening. The script then can overload the *dispatch()* function of the instantiated controller, to execute arbitrary code when events occur for that controller.

Here below an example of this use with the most basic controller: the *TriggerController*, once created and registered, will call dispatch() every time a frame is processed:

```
import threading
import freej

# context and screen initialization
cx  = freej.Context()
scr = freej.SdlScreen( 400, 300 )
cx.add_screen( scr )


### declare the Trigger Controller
class Frame(freej.TriggerController):

  def __init__(self, *args):
    super(Frame, self).__init__(*args)


  ### the dispatch function is the callback
  ### it will be called at every frame

  def dispatch(self):
      ### rotate around 360 degrees, incrementing
      ### this function is called once every frame
      if self.i>360: self.i=0
      self.i += 1
```

```
        self.txt.set_rotate( self.i )
        return 1
        # dispatch should always return an integer value



### create an instance of our Trigger Controller
f = Frame()
### set rotation index to zero
f.i = 0
### create a text layer inside the controller
f.txt = freej.TextLayer()
f.txt.init(cx);
f.txt.write("Hello World!")
f.txt.start();
cx.add_layer(f.txt);


# register it on the current context
cx.register_controller(f)


# start running freej in a separate thread
th = threading.Thread(target = cx.start , name = "freej")
th.start();
```

It initializes a new FreeJ instance as usual, then declares the new class *Frame* overloading the TriggerController. With this new class we overload the *dispatch* function to print out some output every time is called, that is every frame.

At last the script creates an instance of the new Frame class (inheriting all methods from TriggerController) registers it to be called by the engine with the *register_controller()* method in the Context.

Once initialized this simple script will call our dispatch function, executing all its contents, every time a frame is processed: so far that is every time that cafudda() is called.

Chapter 6. Ruby

There are bindings to use FreeJ in Ruby, these are at an early stage of development, actually ready for the first explorers :) make sure you installed FreeJ complete with Swig to wrap Ruby bindings

Hello World

Here below the classical "Hello World" script that simply opens up a FreeJ output window and a Text Layer, writing into it,

After initialisation of *cx* as a FreeJ context a *TextLayer* is initialised and added to it. Please note the sequence for creating the layer is fixed: first the constructor, then *init()* is called passing the context as an argument, some text is written into it using the *write()* method, the layer is then started and added to the context.

Play and apply an effect

The following script should be run with an argument: an image or video file that will be reproduced with a filter effect applied on it.

```
require 'Freej'

cx = Freej::Context.new
cx.init(400,300,0,0)

cx.plugger.refresh(cx)

lay = Freej::create_layer(cx,ARGV[0])

filter = cx.filters.search("vertigo")[0]
lay.add_filter(filter)

cx.add_layer(lay)

# main loop
while 1
  cx.cafudda(0.0)
  sleep 0.04
end
sleep 10
```

This script creates and initialises a context, that is an instance of FreeJ, with a size of 400 by 300; the size can also be adjusted interactively later, with screen implementations that support it. It then completes initialisation of plugin effects refreshing the list of those present on the system using *plugger.refresh*.

It then opens the file given at commandline as first argument (argv) and starts it in a layer, adds the visual effect "vertigo" to the layer created, adds the layer to the screen and runs.

However, the above example is not all you can already do with FreeJ: all the C++ API is mapped to various languages and you can use it as it is done in its code, which right now is the main documentation source, already reasonably documented via doxygen. For example, studying the header of the video_encoder implementation and how it is used in the C++ code, you can recognize the classes that are accessible from Python and script the encoding and streaming capabilities. There are many possibilities out there ;^)

Controllers

Controllers in FreeJ are listeners which will dispatch asynchronous messages from input channels, that is for example a keyboard that will activate callbacks every time keys are pressed, informing the running script that something is happening. The script then can overload the *dispatch()* function of the instantiated controller, to execute arbitrary code when events occur for that controller.

Here below an example of this use with the most basic controller: the *TriggerController*, once created and registered, will call dispatch() every time a frame is processed:

```
require 'Freej'

cx = Freej::Context.new
cx.init(400,300,0,0)

class Frame < Freej::TriggerController

      def dispatch

         puts("dispatch callback activated")

# please  note  the following return: is needed
# at  the end of the function, else Ruby will
# return an exception and bail out

     return(1)

      end
end

f = Frame.new
cx.register_controller(f)
```

It initializes a new FreeJ instance as usual, then declares the new class *Frame* overloading the TriggerController. With this new class we overload the *dispatch* function to print out some output every time is called, that is every frame.

At last the script creates an instance of the new Frame class (inheriting all methods from TriggerController) registers it to be called by the engine with the *register_controller()* method in the Context.

Once initialized this simple script will call our dispatch function, executing all its contents, every time a frame is processed: so far that is every time that cafudda() is called.

Index