The GePhex Book

Martin Bayer

Georg Seidel

The GePhex Book

Martin Bayer
Georg Seidel
Copyright © 2002, 2003, 2004, 2005 Martin BayerGeorg Seidel

Abstract

GePhex is an interactive effect system for video jockeys. The effects can be controlled with external devices like joysticks, web-cams, or midi-devices. New effects can be designed in a GUI (Graphical User Interface) by composing basic effects into more complex ones.

This book gives new users an introduction to the GePhex system. The basic concept of effect-graphs is described. The reader learns how to use the system and the steps to create new effects. The last chapters hold information for developers.

This document is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but *WITHOUT ANY WARRANTY*; without even the implied warranty of *MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE*. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Table of Contents

1. Introduction 1
    What is GePhex 1
    GePhex's History 1
    Security Note 2
    GePhex's Components 2
        The Engine 2
        The GUI (graphical user interface) 4
        The GePhex Script 5
2. Installation 6
    Building from the Sources 6
        Getting the latest Version from the Arch Archive 6
        Getting a Distribution-Tarball 6
        Building on Unix Platforms 7
        Building on Mac OS X 10
        Building on Windows 11
    Precompiled Versions 13
        Using our APT Repository 13
        Windows binary Distribution 14
3. Guided Tour 15
    Starting GePhex 15
        UNIX 15
        Windows 15
    The GePhex Graphical User Interface 15
        Structure of the GUI 15
        The First Graph 16
        Adding Effects to the Graph 17
        Configuring the Graph 19
        Running and controlling the Graph 19
        Saving the Graph 20
4. Basic Concepts 21
    The three States of Graphs in the Renderer 21
5. Example Graphs 22
    Example1: Tunnel-Vision 22
    Example2: Plasma 22
    Example3: A simple Feedback Loop 22
6. Module Reference 23
    Generators 23
        ifsmodule 23
    Input Modules 23
        Video-playback (avifilemodule) 23
        Video for Linux (capturemodule) 24
7. Type Reference 26
    NumberType 26
    FrameBufferType 26
8. Developer Information 27
    Adding new data types 27
        The c-API 27
        An example for a new data type 39
    Adding new effect modules 41
        The C-API 41
        Pluc the skeleton generator 56
        An example for a new module 59

List of Tables

1.1. Options for gephex-engine 2
1.2. Options for gephex-gui 4
8.1. Most important pluc commands 56
8.2. Mandatory global settings 57
8.3. Mandatory input settings 57
8.4. Optional input settings 58
8.5. Mandatory output settings 59

Chapter 1. Introduction

What is GePhex

GePhex is a software-based interactive video-effect system. Video jockeys can use this system to modify or recombine existing footage or create new video effects in an interactive process. External devices like joysticks, midi-keyboards, or web-cams can influence the real-time video generation.

- GePhex is Free Software. You can use, distribute and modify GePhex under the terms of the GPL.
- GePhex is a multi platform project. Supported operating systems are Win32, Linux, and Mac OS X. There is also a port to FreeBSD.

GePhex allows the construction and modification of video-effects on different levels:

- The users view of a video-effect is a data-flow graph with sources, modifiers and destinations. The data in these effect-graphs are typed. That means the inputs and outputs of the modules have types e.g. video, color or number. The user can create complex effects by connecting inputs and outputs of the same type.
- Not all inputs must be connected. The user can set the values for unconnected inputs with the GUI. These values can be saved in so called snapshots.
- The effects can be influenced by the environment in two ways. First, special source-modules inject data from hardware devices like midi-devices, web-cams, or joysticks. Second it is also possible to connect special GUI elements to the inputs of a module (think slide-bars, color choosers, file dialogs, ...).
- Developers can extend the system with the plugin mechanism for modules and types.

GePhex's History

- The project started in the late summer 2001.
- In autumn 2002 was the first public vjing session at the fmi party on the campus of the university of Passau, Germany.
- In autumn 2003 there was a public session in Vienna and another session at the fmi party in Passau, Germany.
- In December 2003 we released the first stable version (0.0.4).
- In June 2004 we released the second stable version (0.4). At this time we also started with the design of version 0.5, which will be a complete rewrite. As of the time of this writing (March 2005), there is only a minimal prototype system of gephex-0.5.
- In November 2004, we were invited to the *piksel* [http://www.piksel.org] meeting in Bergen, Norway. We released version 0.4.1 shortly before.

- Version 0.4.2 was released some weeks after piksel. It includes support for *frei0r* [http://www.piksel.org/Frei0r], a simple effect API we designed in Norway, and some other stuff we did there (e.g. v4l2 support).
- In February 2005, experimental support for Mac OS X and ports of most *effecTV* [http://effectv.sf.net] effects was added to version 0.4.3.

Security Note

With the default configuration the GePhex engine listens at the TCP port 6666. The GUI connects to this port to control the engine. There is no authentication necessary to connect to the engine. This could be a security hole if used in a hostile environment. Never start this software as root and don't use it in an network that is connected to the internet without protection (e.g. a firewall).

GePhex's Components

The Engine

The engine can be started from the console with the `gephex-engine` command. The only environment variable that is used is `DISPLAY` for the output under X11. All options are set in the configuration file `~/.gephex/0.4/gephex.conf`. If this file does not exist, a default configuration file is created.

```
common {
    media_path           = [/home/tmp/seidel/gphx/share/gephex]
}

engine {
    module_path          = [/home/tmp/seidel/gphx/lib/gephex/modules/]
    type_path            = [/home/tmp/seidel/gphx/lib/gephex/types/]
    frei0r_path          = [/home/cip/seidel/.frei0r-1/lib/]
    graph_path           = [/home/cip/seidel/.gephex/0.4/graphs/]
    ipc_unix_node_prefix = [/tmp/gephex_socket_]
    ipc_type             = [inet]
    ipc_port             = [6666]
    renderer_interval    = [40]
    net_interval         = [40]
}
...
```

In the following, a `path` is a semicolon separated list of directories, which are searched by the engine in the given order.
`media_path` is used for both engine and gui. It tells the engine and the gui where media (i.e. images, videos, fonts, ...) is located.

Table 1.1. Options for gephex-engine

Option	Explanation	Type
`module_path`	tells the engine where to look for gephex effect modules	string
`type_path`	tells the engine where to look for gephex types	string

Option	Explanation	Type
`frei0r_path`	tells the engine where to look for *frei0r* [http://www.piksel.org/Frei0r] effects	string
`graph_path`	tells the engine the location of the user created graphs. The first directory in this path is used to store new graphs.	string
`ipc_type`	tells the engine how to connect to the user interface. You can select between `inet` for a internet-protocol based communication, on Unix platforms `unix` for Unix domain sockets and `namedpipe` for named pipes on win32 systems. If engine and GUI run on the same host the usage of non ip based communication is preferred because it has less overhead.	string
`ipc_unix_node_prefix`	specifies where the unix node (fifo) is created when using `ipc_type unix`	string
`ipc_port`	specifies the port number when using `ipc_type inet`	int
`renderer_interval`	tells the engine how often the current graph should be updated. It expects the time between updates in milliseconds. A value of 40 is equivalent to 25 updates per second.	int
`net_interval`	tells the engine how often to read commands from the user interface.It expects the time between reads in milliseconds.	int
`autostart`	if set, the engine automatically starts the update loop	bool
`render_graph_id`	the id of the graph that should be loaded at the beginning	string
`render_snap_id`	the id of the snapshot that should be loaded at the beginning	string
`ttl`	time to live: if the value is not 0, the engine terminates itself after `ttl` update cycles	int

All options that can be set in the config file can be overridden from the command line. For example

```
> gephex-engine --ipc_port=1234
```

overrides the value of the option `ipc_port`.

The `--help` option shows a list of available parameters with a short help text.

```
    > gephex-engine --help
                  welcome to...
       ---             _           ---
    ----              | |            ----
  ----   __ _ ___ _ __| |  ___ _  __  ----
 -----  / _' | - \ '_ \ |_/ - \ \/ /  -----
------ | (_| | --/ |_) |  \ --/>  <   ------
 -----  \__. | \_| .__/ |'|\__/ /\ \  -----
  ----   ._' |\__/ |  |_||_|__//  \_| ----
   ----  |__/    |_|                 ----
      ---                           ---

[version:    "0.4.3"]
[build time: "Mar  5 2005 - 00:13:18"]
Usage: /Users/georg/gphx//bin/gephex-engine-real [options]
The allowed options are:
  --media_path  List of directories that contain videos, images and fonts (separat
...
```

The GUI (graphical user interface)

The gui can be started on the console with the `gephex-gui` command. There are two environment variablse that influence the guis behaviour, see the note at the end of the section. All options are set in the configuration file `~/.gephex/0.4/gephex.conf`. If this file does not exist a default configuration file is created.

```
common {
    media_path           = [/home/tmp/seidel/gphx/share/gephex]
}

...

gui {
    ipc_type             = [inet]
    ipc_inet_hostname    = [localhost]
    ipc_port             = [6666]
    ipc_unix_node_prefix = [/tmp/gephex_socket_]
    engine_binary        = [/home/tmp/seidel/gphx/bin/gephex-engine]
}
```

`media_path` is used for both engine and gui. It is a semicolon separated list of directories, which contain media for GePhex (i.e. images, videos, fonts, ...).

Table 1.2. Options for gephex-gui

Option	Explanation	Type
`ipc_type`	tells the engine how to connect to the user interface. You can select between `inet` for a internet-protocol based communication, on Unix platforms `unix` for Unix domain sockets and `namedpipe` for named pipes on win32 systems. If engine and GUI run on the same host the usage of non	string

Option	Explanation	Type
	ip based communication is preferred because it has less overhead.	
`ipc_unix_node_prefix`	specifies where the unix node (fifo) is created when using `ipc_type unix`	string
`ipc_port`	specifies where the number when using `ipc_type inet`	int
`engine_binary`	the full path of the engine executable. It is used to spawn a engine process if necessary.	string

All options that can be set in the config file can be overridden from the command line. For example

```
> gephex-gui --media_path=/mnt/data/footage/
```

overrides the value of the option `media_path`.

The `--help` option shows a list of available parameters with a short help text.

```
    > gephex-gui --help
Usage: /Users/georg/gphx//bin/gephex-gui-real [options]
The allowed options are:
  --media_path  List of directories that contain videos, images and fonts (separat
...
```

The GePhex Script

On unix platforms, a script called gephex is installed. It takes care of starting the engine and the gui.

The engine is started in an x-terminal-emulator, which defaults to x-terminal-emulator, and if that does not exists on your system xterm is used.

To override this behavior, you can set the GEPHEX_XTERM environment variable. Make sure that your terminal emulator accepts the "-e" flag to execute a program. To add extra flags to your terminal emulator, use the GEPHEX_XTERM_FLAGS environment variable. Example:

```
   > GEPHEX_XTERM=aterm GEPHEX_XTERM_FLAGS="-fg green -bg black" gephex
```

If you need even more control, look at the file ~/.gephex/run_in_terminal.sh. If you want to restore the default for some reason, it's OK to simply delete it, gephex will copy the default file if it does not find it there.

Note

The gui uses the same mechanism to start the engine, so you can provide the GEPHEX_XTERM* variables for the gui, too.

Chapter 2. Installation

Since GePhex is free software you can get the source code and compile it on you own. Or if you have one of our core platforms you can download the precompiled binaries from our *website* [http://www.gephex.org].

Building from the Sources

It is not very difficult to compile your own version of GePhex. Especially on Unix, if you already installed software with "configure" and "make install" there should be no big surprises for you here. On windows, you need a copy of ms visual studio 6 and several libraries.

Before we can start we must get a version of the GePhex source code. There are official releases as tar balls on the *website* [http://www.gephex.org/download.php] and the developer versions available via *arch* [http://www.gnu.org/software/gnu-arch/].

Getting the latest Version from the Arch Archive

I assume that you use the tla arch-client. To get the latest version you need to register the gephex archive. The name of the archive is `gephex@gephex.org--2004` and the location is `http://arch.gephex.org/gephex/2004`.

```
bash@host:~$ tla register-archive http://arch.gephex.org/gephex/2004
Registering archive: gephex@gephex.org--2004
bash@host:~$
```

The full name of the revision is `gephex--main--0.4`:

```
bash@host:~$ tla get -A gephex@gephex.org--2004 gephex--main--0.4 GePhex
* from archive cached: gephex@gephex.org--2004/gephex--main--0.4--patch-1727
* patching for revision: gephex@gephex.org--2004/gephex--main--0.4--patch-1728
* making pristine copy
* tree version set gephex@gephex.org--2004/gephex--main--0.4
bash@host:~$
```

This command will fetch the latest version of gephex into the directory `./GePhex`.

If you have questions regarding arch please have a look at the projects *homepage* [http://www.gnu.org/software/gnu-arch/].

Getting a Distribution-Tarball

Visit our *download page* [http://gephex.org/download.php] and get a release. Let's assume the tar ball is called `gephex-0.4.tar.gz`. Just unpack the file.

```
bash@host:~$ tar xvzf gephex-0.4.tar.gz
gephex-0.4/AUTHORS
gephex-0.4/BUGS
gephex-0.4/ChangeLog
gephex-0.4/INSTALL
...
```

On windows you can use, for example, *WinRAR* [http://www.rarlab.com] to unpack the file.

Building on Unix Platforms

Bootstrapping

This step is only necessary when you got the sources from the arch repository. The distributed tar balls are already "bootstrapped".

To do the bootstrapping, you need some additional software. This includes autoconf and automake and some additional packages to build the documentation etc. You should use rather new versions of automake and autoconf. For automake, 1.8 is known to work and 1.4 is known to *not* work with our build system. For autoconf, 2.59 is known to work.

Note

When installing from a tar ball, this software is *not* needed!

- *autoconf* [http://www.gnu.org/software/autoconf/] *automake* [http://www.gnu.org/software/automake/] *libtool* [http://www.gnu.org/software/libtool/]
- *python* [http://www.python.org/]
- *DocBook DTD* [http://docbook.sourceforge.net/projects/docbook/] *DocBook XSL Stylesheets* [http://docbook.sourceforge.net/projects/xsl/] *xsltproc* [http://xmlsoft.org/XSLT/] *docbook2x* [http://docbook2x.sourceforge.net]

GePhex uses autoconf and automake for the configuration and generation of Makefiles. Use the script `bootstrap.sh` to create the build-system without further intervention.

```
bash@host:~$ cd GePhex
bash@host:~/GePhex$./bootstrap.sh
running aclocal ...
running libtoolize --force ...
running autoheader ...
running automake --add-missing --copy ...
running autoconf ...

./configure has been successfully built!
See './configure --help' for available options
```

Configure and Build

This step configures the build system for your system. You need the following libraries to get all the features of GePhex:

- *libqt* [http://doc.trolltech.com/3.3/index.html] *xlib* [http://www.xfree.org/]
- *libsdl* [http://www.libsdl.org/index.php] *libpng* [http://www.libpng.org/pub/png/libpng.html] *mesa* [http://www.mesa3d.org/] *avifile* [http://avifile.sourceforge.net/] *alsa* [http://alsa-project.org/] *aalib* [http://aa-project.sourceforge.net/aalib/]

At the very least you should have libqt and xlib installed. If not the GUI will not be built.

At this point you can choose the location where the software should be installed. Some special options to include/exclude features can also be activated here. Normally it is not necessary to use any special parameters because everything is checked automatically by the configure script.

To see the available options you can use the `configure` script:

```
bash@host:~/GePhex$./configure --help
`configure' configures this package to adapt to many kinds of systems.
...
Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
...
Optional Features:
...
  --enable-mmx            Turn on MMX support (still runs on x86 that don't
                          have MMX) [default=yes]
  --enable-serialize-framebuffer
                          Serialize the framebuffer type (for previews in the
                          gui) [default=no]

 Optional Packages:
...

  --with-SDL_IMAGE        Turn on SDL_IMAGE support (default=yes).
  --with-SDL_TTF          Turn on SDL_TTF support (default=yes).
  --with-AVIFILE          Turn on AVIFILE support (default=no).
  --with-MPEG3            Turn on MPEG3 support (default=no).
  --with-LIBPNG           Turn on LIBPNG support (default=no).
  --with-AALIB            Turn on AALIB support (default=yes).
  --with-ASOUNDLIB        Turn on ASOUNDLIB support (default=yes).
  --with-V4L              Turn on V4L support (default=yes).
  --with-OSS              Turn on OSS support (default=yes).
  --with-LINUX_JOYSTICK   Turn on LINUX_JOYSTICK support (default=yes).
  --with-FFMPEG           Turn on FFMPEG support (default=yes).

Some influential environment variables:
...
  FRBINCACHE  The size of the cache of the image source module in MB
...
```

So let's do the actual configuration:

```
bash@host:~/GePhex$ ./configure --prefix=/usr
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for g++... g++
...
```

Now we can start the build process. Depending on you system this could take a long time.

```
bash@host:~/GePhex$ make
```

```
make  all-recursive
make[1]: Entering directory `/home/martin/code/gephex/GePhex'
Making all in base
make[2]: Entering directory `/home/martin/code/gephex/GePhex/base'
Making all in src
make[3]: Entering directory `/home/martin/code/gephex/GePhex/base/src'
...
```

Installation

The following command installs the software on your system. The two binaries for the user interface `gephex-gui` and the rendering engine `gephex-engine` and the gephex wrapper script `gephex` will be installed in the `PREFIX/bin` directory and the location of the plugins is in `PREFIX/lib/gephex`.

```
bash@host:~/GePhex# make install
```

Note

You might need to be root to install (depending on the installation prefix you chose).

Create your own Debian Packages

Users of the Debian GNU/Linux OS can create Debian binary packages from the source and install these as root with dpkg.

To build Debian packages you need these Debian packages installed:

- debhelper
- libqt3-mt-dev
- libsdl1.2-dev
- xlibmesa-dev
- libpng12-dev
- libavifile-0.7-dev
- libsdl-ttf2.0-dev
- libasound2-dev
- libmpeg3-dev
- libsdl-image1.2-dev
- automake1.8
- libtool
- autoconf

- python
- docbook-xml
- docbook-xsl
- docbook2x
- xsltproc

Not all necessary files for building the Debian packages are included in the dist tar balls. You need to get a full copy of the gephex source tree from the arch repository. The necessary steps to do this are described above.

First bootstrap the clean source tree by calling the bootstrap.sh script in the root of the source tree. Then build the packages with the dpkg-buildpackage commando. After a successful build you can install the generated packages as superuser with dpkg.

```
bash@host:~/GePhex$ ./bootstrap
bash@host:~/GePhex$ fakeroot dpkg-buildpackage
bash@host:~/GePhex$ sudo dpkg -i ../gephex_0.4-1_i386.deb
```

Building on Mac OS X

Building on Mac OS X is very similar to building on other UNIX systems.

Note that the tools described here are only one choice of several alternatives. If you find a better way to build gephex on Mac OS X *let us know* [http://lists.gephex.org/mailman/listinfo].
Here comes a list with tools that can be used to build gephex.

- *fink* [http://fink.sourceforge.net/]
- the developer tools from apple (I got them from my installation cd, but i installed an update to gcc from *here* [https://connect.apple.com/])
- x11 server + sdk (for example the sdk for *the x11 server provided by apple* [http://www.apple.com/macosx/features/x11/] comes as an extra package with the developer tools)
- qt (I used finks qt)
- sdl and sdl_image (for loading images other than bmp files, again I used finks versions)

After that everything should work as described in the section called “Building on Unix Platforms”.

If you manage to compile and link gephex against the native qt version for Mac OS X please *let us know* [http://lists.gephex.org/mailman/listinfo]!
We are looking for people to improve gephex on Mac OS X. Important areas that need work:

- audio support (audio input and audio output module)
- better output driver (current x11 output is slow, sdl seems broken)
- altivec optimisations of some basic modules (xfader, ...)

If you are interested *let us know* [http://lists.gephex.org/mailman/listinfo]!

Building on Windows

What you need

The following programs and libraries are needed to compile gephex on win32:

- ms visual studio 6 (at least the command line interpreter)
- *python* [http://www.python.org/download/]
- *nasm* [http://nasm.sourceforge.net/]
- *libqt (2.3nc)* [http://www.trolltech.com/download/qt/download_noncomm.html]
- *sdl* [http://libsdl.org]
- *sdl-image* [http://www.libsdl.org/projects/SDL_image]
- *sdl-ttf (choose the devel zip file for visual studio 6)* [http://www.libsdl.org/projects/SDL_ttf]
- *directx sdk* [http://www.microsoft.com/windows/directx/default.aspx]
- *ffmpeg (distributed with GePhex in source form)* [http://ffmpeg.sourceforge.net]
- *MinGW/MSYS (needed to compile ffmpeg)* [http://www.mingw.org]

Environment and Path

The python, nasm and qt (uic, moc) binaries must be in the PATH. You need to set up the following environment variables:

SDL_DIR	base dir of SDL
SDL_DIR	base dir of sdl
SDL_IMAGE_DIR	base dir of sdl-image
SDL_TTF_DIR	base dir of sdl-ttf
QTDIR	base dir of qt
DXSDK_DIR	base dir of directx sdk

Building ffmpeg

Before building GePhex, ffmpeg must be built. Fire up MSYS and chdir into "${gephex-dist}/contrib/ffmpeg". Then execute

```
> ./configure --enable-shared
> make
```

If everything works out, the necessary dlls are now in "${gephex-dist}/contrib/ffmpeg/libavformat" and

"${gephex-dist}/contrib/ffmpeg/libavcodec". There are two ways to build GePhex (all paths are relative to your gephex dir):

Building with the Visual Studio IDE

Create the following empty folders in your gephex dir:

- `$gephex-dir/dlls`
- `$gephex-dir/dlls/modules`
- `$gephex-dir/dlls/types`
- `$gephex-dir/graphs`

Copy

- `$gephex-dir/data/gephexw.conf.default` to `$gephex-dir/gephex.conf`,
- `$gephex-dir/config_h.win32` to `$gephex-dir/config.h`, and
- the graphs in `$gephex-dir/examples/graphs` to `$gephex-dir/graphs`.

Fire up vs6 and open `$gephex-dir/build/gephex.dsw`. Choose your configuration (Release/Debug). You have to build three projects:

- engine
- gui
- dummy (builds all modules and types)

Building via the Command Line

Note

Unfortunately, vs6 puts hard paths into .mak files and .dep files. Although the dsp files are controlled by the above environment variables, the console build will probably only work if you have installed the gephex source and your libraries into the following directories:

- `GEPHEX_DIR` = `c:/code/gephex-0.4` (and _not_ `c:/code/gephex-0.4.x`!)
- `SDL_DIR` = `c:/code/sdl`
- `SDL_IMAGE_DIR` = `c:/code/sdl_image`
- `SDL_TTF_DIR` = `c:/code/sdl_ttf`
- `QTDIR` = `c:/code/qt`
- `DXSDK_DIR` = `c:/dxsdk`

Additionally, you have to rename your gephex dir to `C:\code\gephex-0.4`. (If anybody knows a better way to create the .mak and .dep files, please tell us!)
The batch file `$gephex-dir/build/make_all.bat` should build everything.

```
> cd build
> make_all Release
```

Note

nmake seems to crash when called from inside cygwin. Use the windows command line instead.

Making it work

Wether you built GePhex with the IDE or with the CLI, you need to make sure that the dlls of the libraries are found when gephex is started. The easiest thing is to put them into $gephex-dir/bin. Alternatively, you could put the dlls into a system path, like c:\windows. The following dlls are needed:

- sdl.dll
- sdl_ttf.dll
- sdl_image.dll
- jpeg.dll
- libpng1.dll
- zlib.dll
- qt-mt230nc.dll
- avcodec.dll
- avformat.dll

Instead of copying these files manually, you can put them into C:\gphx_dist_data\extra_dlls. Then, you can use the script $gephex-dir/build/make_binary_dist.bat to create a working binary distribution.

```
> cd build
> make_binary_dist C:\my_gephex_0.4_dist
```

Note

You need the recode utility, available with the *UnxUtils* [http://unxutils.sourceforge.net/] for this batch file.

Precompiled Versions

Using our APT Repository

The Debian binary package format is the common way to install software on a Debian GNU/Linux system. Using the dpkg, a medium-level tool to install, build, remove and manage Debian GNU/Linux packages, the system stays in a consistent state after changes to the software installation or configuration. The proper deinstallation and upgrade to a other version of the application package is guaranteed.

Add the GePhex apt repository lines to your /etc/apt/sources.list and install GePhex with apt-get:

```
bash@host:~$ sudo cat >> /etc/apt/sources.list
deb http://www.gephex.org/debian/ unstable main
bash@host:~$ sudo apt-get install gephex
```

Windows binary Distribution

Just get the windows binary .rar or .zip file and unpack it. GePhex can be started from the `bin` directory.

Chapter 3. Guided Tour

Starting GePhex

UNIX

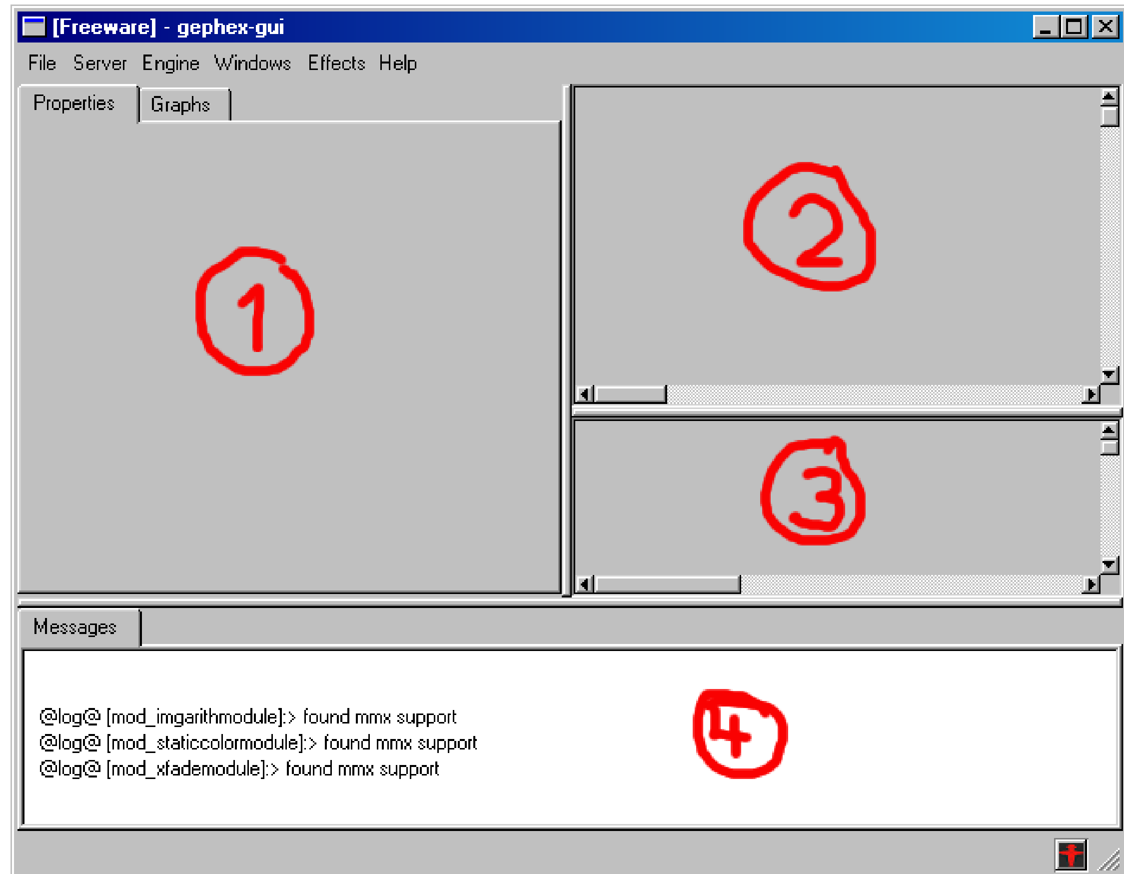
You can start GePhex by executing `gephex`.

If you chose your own prefix for the install, make sure the gephex binary is in the path.

Windows

Just go into the bin directory of your GePhex directory and execute the gephex-engine and gephex-gui (in that order).

The GePhex Graphical User Interface



The GUI (just started).

Structure of the GUI

As you can see in the image, the GUI (Graphical User Interface) is divided into four major areas. The

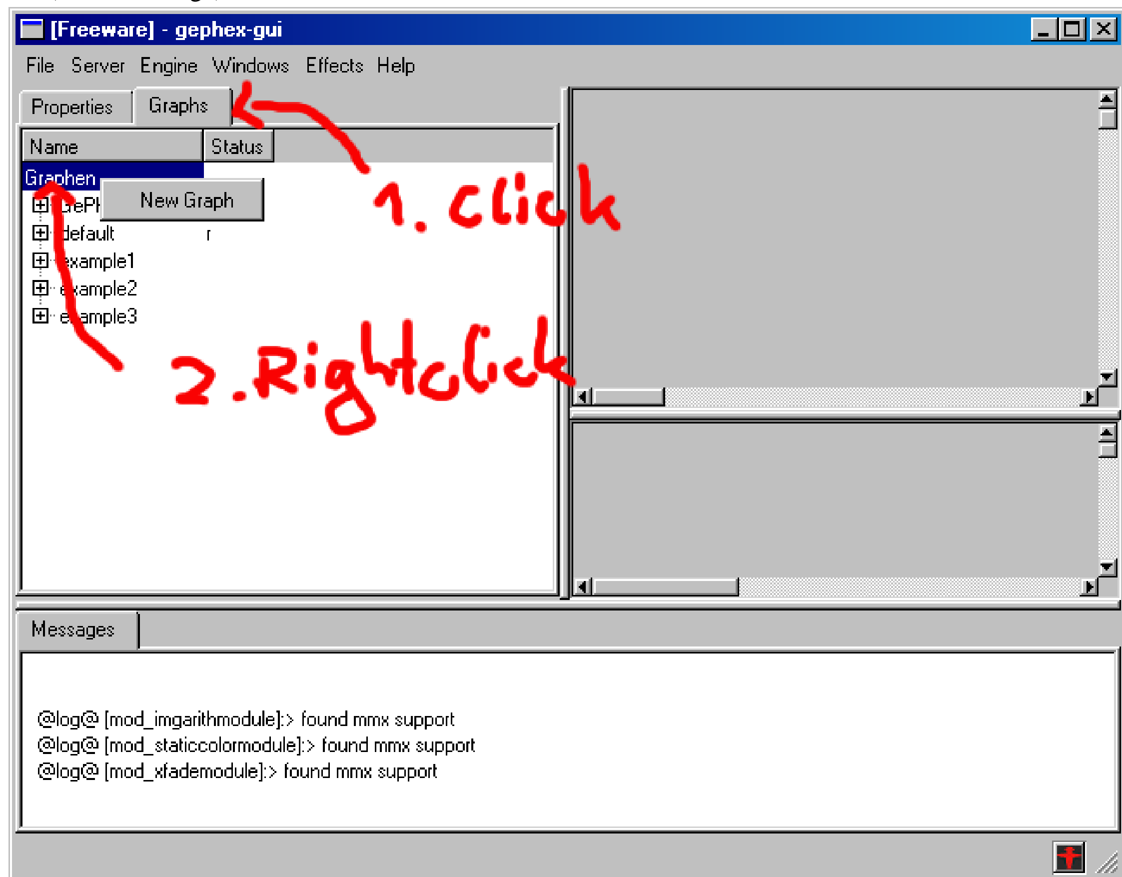
areas are marked with a red pen.

- 1: Info-window. Used for properties of effects and for loading and saving effect-graphs,
- 2: Graph-window. Used to edit effect-graphs.
- 3: Control-window. Used to control running effect-graphs.
- 4: Message-window. Used to display error and warning messages.

The First Graph

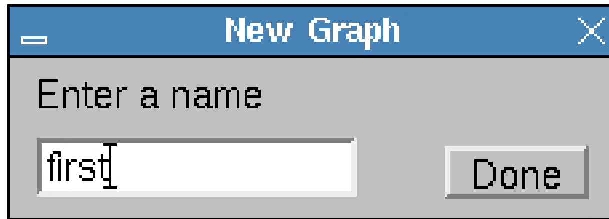
The most important concept for using GePhex is that of an effect-graph. An effect-graph is a number of simple basic effects, combined to perform a more complex effect.

Since GePhex works with graphs, we have to tell it which graph we want to edit. Do this by clicking on the "Graphs" tab in the info-window. To create a new graph, right-click on the "Graphs" item inside the tab (see next image).



Creating a graph

When the context-menu opens up, just select "New Graph". Enter "first" in the dialog.



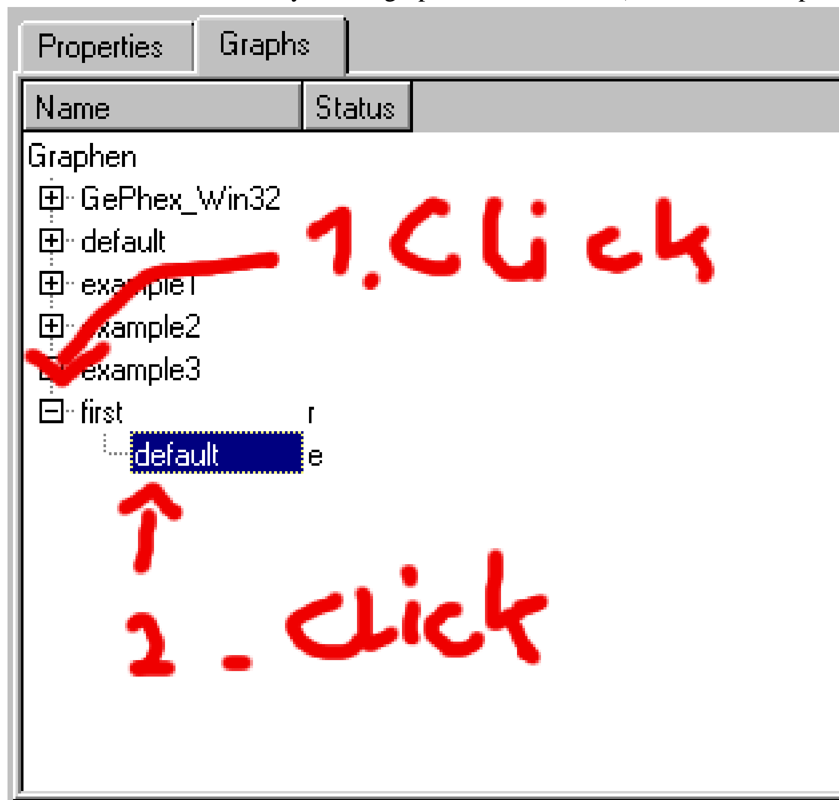
Choosing the name of the graph

Now you see another item called "first" in the tree-view below "default". This is our new graph. To activate it, click on the arrow (or plus symbol) left to "first". Click on the appearing child item "default".

Note

This child item is a "snapshot" of the graph. For now you just need to know, that you need one active graph with one active snapshot in order to create an effect.

The letters "r" and "e" tell you that graph "first" is active (with current snapshot "default").

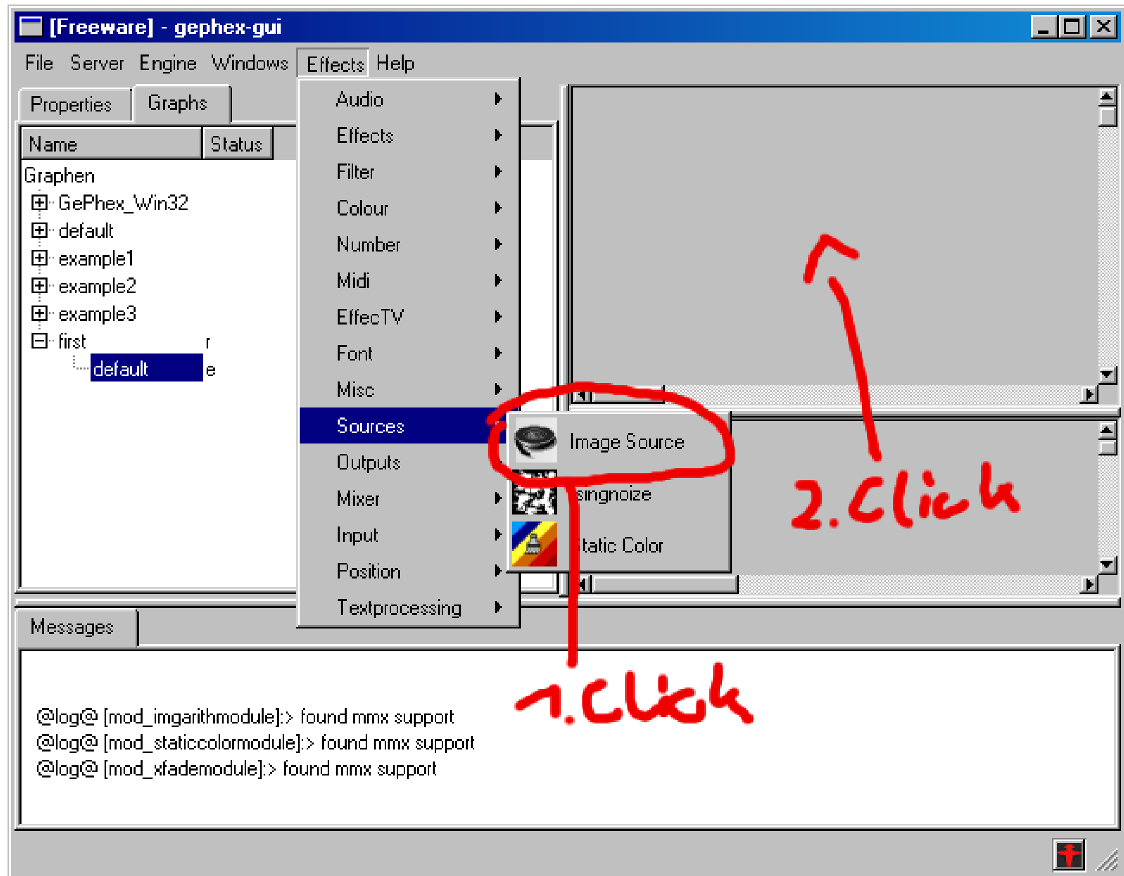


Activating the graph

Adding Effects to the Graph

Now we have created a new graph. But it is not very useful yet, because it is empty. So let's create some effects.

To do so, open the "Effects" menu in the top-level menu-bar. Choose "Sources"->"Image Source". Then click into the graph window as shown in the next image.

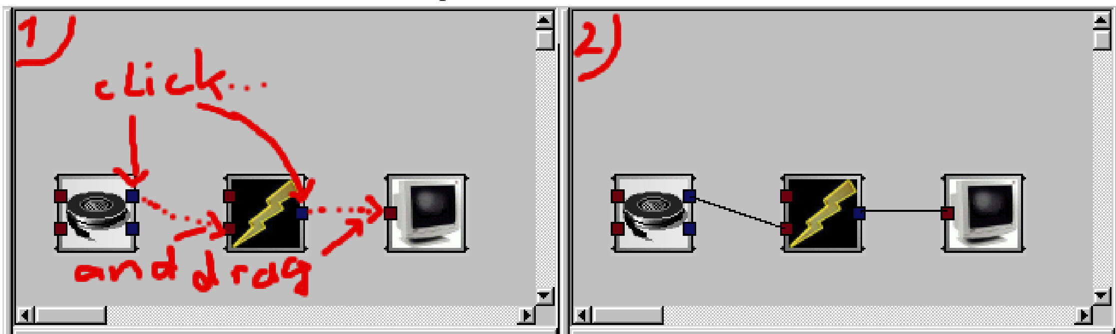


Adding an basic effect

Do the same for "Effects"->"Outputs"->"Image Output" and "Effects"->"Filter"->"FlashFader". Arrange them as in the next image (You can simply move them around with the mouse).

The red boxes on the left side of the basic effects are inputs, the blue buttons on the right are outputs.

Connect the effects as shown in the next picture:



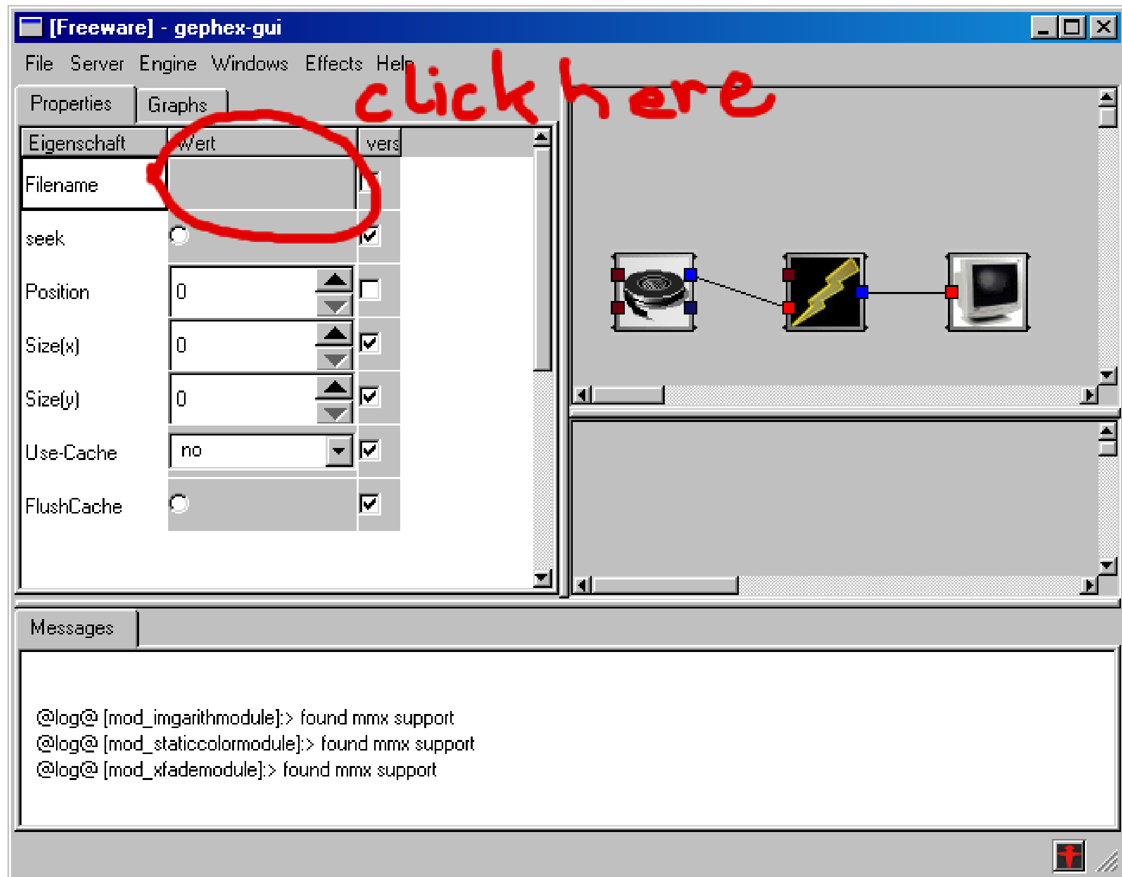
Connecting effects

The graph we have created so far is still very simple. In fact, you would not call such a graph an effect at all. What it does is simply loading a bitmap, eventually flashing it and displaying it to the screen.

You can think of this graph in terms of data flow: data comes from a source (the image source), flows through the flash-fader filter and is displayed in the sink (the image output).

Configuring the Graph

This is simple. We just choose a bitmap. Right-click on the "Image Source"-effect and choose "Properties" in the context menu. Now the info-window displays the properties of the image source:



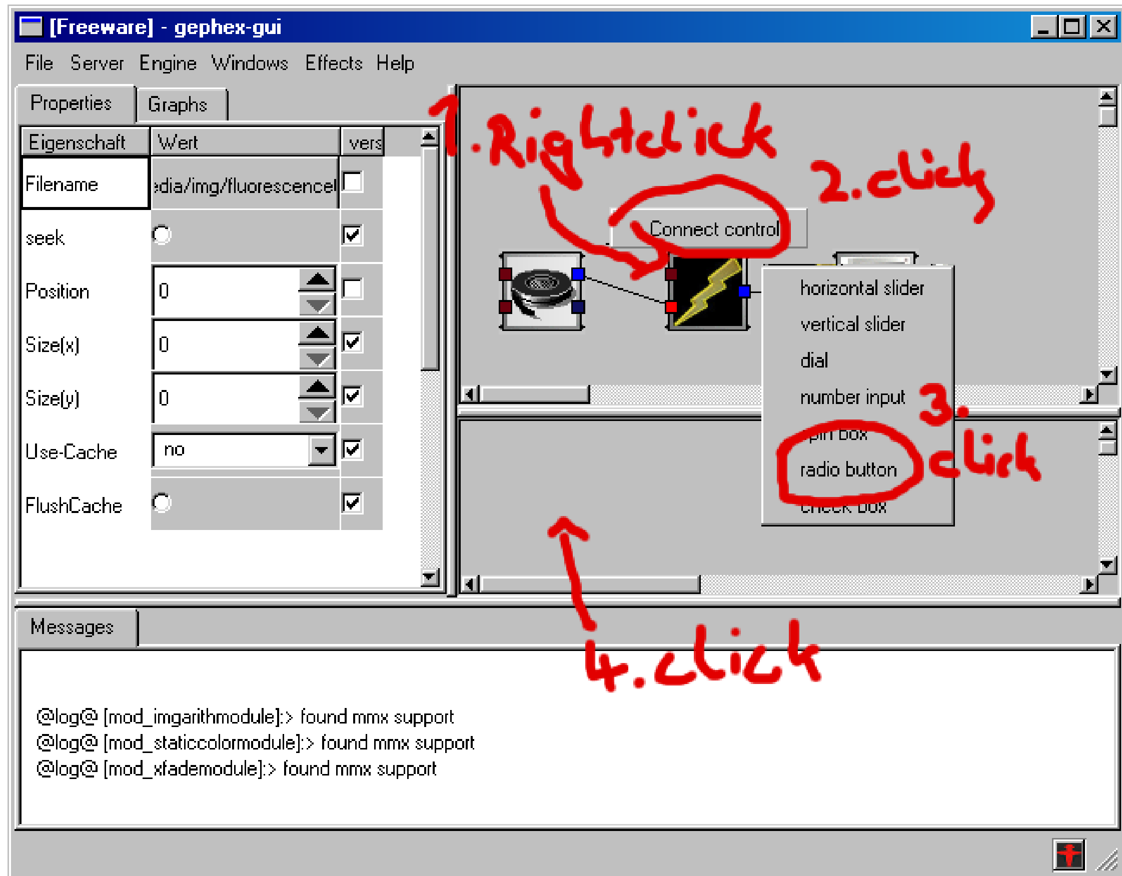
Properties

Click on the button next to "Filename" and choose a nice image file.

Running and controlling the Graph

Simple again. Just click on the little red fellow on the bottom right. You should see the output window opening and displaying the bmp file you chose. To inform you that it is running, GePhex turns the red fellow to a green fellow.

To control the flash-fader effect we must add a control at the upper input of the flash-fader. The following picture explains how you can add a control to an input:



Creating a control

Just try it!

Note

GePhex must be running for the control changes to take effect!.

Saving the Graph

Click on the "Graphs" tab in the info-window. Right-click on "first" and choose "Save Graph". Done. The graph will be already there when you start next time.

Chapter 4. Basic Concepts

TODO: this chapter needs a lot of work

The three States of Graphs in the Renderer

The Renderer knows three states for a graph:

- The graph is not loaded. No internal state of the modules is stored and obvious no calculation is done in this state.
- The graph is loaded in the renderer. The modules remember their internal states e.g. the framebuffer of an xfader with loopback. But no calculation is allowed in this state.
- Only in the state active are any calculations done.

So how can I distinguish these states? How can I change between them? What do they imply for the usage of gephex?

Chapter 5. Example Graphs

On UNIX systems, the examples are installed automatically when you run GePhex for the first time. If not, it might be necessary to remove the `~/.gephex/0.4` directory from an older version of GePhex.

Note

This could delete your graphs, so backup `~/.gephex/0.4/graphs` if necessary.

Example1: Tunnel-Vision

Shows how to use the tunnel.

Try to attach the frbinmodule (Image Source) module instead of the isingnoize.

Example2: Plasma

Simple plasma effect graph.

Example3: A simple Feedback Loop

Try to change the zoom and rotation of the rotozoom-module. If you choose the right parameters, it should look like if you film a monitor that displays what you film...

Chapter 6. Module Reference

Generators

ifsmodule

Linear iterated function systems are a fractal type. The module renders these kind of ifs parameter sets to a image.

This IFS fractal is rendered in gray scale mode.

Input Modules

In this section all modules are listed, who's main goal is to inject data in from external sources in signal graph.

Video-playback (avifilemodule)

Description

There are different video file-formats. Some can be streamed via net. Others allow random access to each video frame. For some there is a normative standard. mpeg 1,2 and 4 are an example for these kinds. avi, quicktime or real video are (re)defined by their vendors. In most cases the video format is just a wrapper for a video stream encoded with a concrete videocodec.

The avifile library extracts the compressed video data from the file formats and and provides with the help of its plugins a lot of codecs to decode the frame sequences. The actual support for one format depends on compile time options and the existence of other libraries on your system. Further information is provided at the homepage of the *avifile* [http://www.avifile.sourceforge.net] project.

Inputs

The first input is the name and path of the video file.

There are two way to control the playback-position. If the seek input is false the module plays the film sequential frame by frame. The playback starts at the the beginning and plays the sequence once. If the seek input is true the playback-position is controlled by the position input. A zero means jump to the beginning and a 1 to the end. If a signal generator is connected to the position the film can be played in reverse, faster or slower just depending on the parameters of the generator.

Outputs

The first output is the video-stream.

The second output is the playback-position in the stream. If the seek is active this is the same as the seek-position but if we disabled seeking this position follows the playback. This output enables looping of parts or setting breakpoints at an arbitrary position.

Notes

Many codecs don't allow fast seeking to an arbitrary video position. This isn't a problem for standard video playback applications. User of a video effect systems want to reverse the playback direction and jump to an random position in the video-footage. Sequential playback is boring. Watch you favorite movie in sine waves!

For random access to the video footage it is often necessary to re-encode the material to frame-based codecs like mjpeg. Tools like *ffmpeg* [http://ffmpeg.sourceforge.net/index.php], *mencoder* [http://www.mplayerhq.hu/homepage/design7/news.html], or *virtual dub* [http://www.virtualdub.org/] are very helpful for these tasks.

Video for Linux (capturemodule)

Description

It is possible to attach several different video-input devices to the computer. Video signals from analog camcorders or vcrs are typically injected by a frame-grabber adapter on the PCI-bus. Digital cameras or low cost web-cams can be connected via USB(2) oder Firewire.

Most operating systems with multimedia capabilities provide a convenience layer between the video-device drivers and the application. All devices are handled independent of the connection type in a similar fashion.

Video4Linux (V4L) is the video capture/overlay API of the linux kernel. It is based on the programming interface introduced by the bttv driver. This is a consumer frame-grabber chip used in most tv cards.

In the 2.6 linux kernel series this api was be replaced by the successor *Video for Linux Two* [http://bytesex.org/v4l/spec/]. The capturemodule works with both versions.

Inputs

The first input is the number of the video device. A setup with a web-cam and a bttv card e.g. uses the device-files /dev/video0 and /dev/video1. You have to set 0 or 1 at this input.

It is possible to switch the device during rendering. But with some hardware/driver combinations this results in one or two broken frames.

The other two inputs tell the frame-grabber the image resolution. A resolution 0,0 forces the grabber to choose any supported resolution.

Output

The captured frames are sent to the output. In case of an invalid video-device, unsupported image size or any other error a black, full transparent one pixel sized image is returned.

Notes

The current version of this module is tested with the 2.4.20 kernel drivers of the the *usb web-cam PCVC740K "ToUCam Pro"* [http://www.smcc.demon.nl/webcam/] from Phillips and the *pci bttv848 frame-grabber card win-TV radio* [http://bytesex.org/bttv/].

Further informations about video4linux driver- and user-space programming can be found in the kernel documentation (kernel-source-2.4.20/Documentation/video4linux/API.html and kernel-source-2.4.20/Documentation/DocBook/videobook.tmpl) and in the *video4linux mailing list* [https://listman.redhat.com/mailman/listinfo/video4linux-list].

Chapter 7. Type Reference

NumberType

64 bit IEEE floating-point value.

FrameBufferType

32-bit BGRA Framebuffertype. Orientation is top-down (topmost line is first line in memory).

Chapter 8. Developer Information

The last chapter described the core effect-modules and data-types included with the GePhex package. None of these are hard-coded in the GePhex engine. All of them are plugins that are loaded at startup time.

This part describes the design for the plugin interfaces.

The main design goal for these two interfaces was simplicity. It should be possible for a programmer to create a new effect within hours and not days. For the modules there exists a code generator which generates Makefiles, stub code and templates to free the programmer from cut and copy operations.

Adding new data types

The focus of the GePhex Framework are streams of video-data. The video streams flow from video sources to the output sinks. To control this flow other types of streams are needed. From simple numbers for controlling the video mixers to complex data types for some special effects, we need different types in the data-flow graph.

The system must be extensible to audio, color-palettes and whatever will be interesting in the future. The types in GePhex are just plugins. You can extend the gephex system with support for new types by providing a shared library, that exports the implementation of a special c-API. The GePhex system loads this library at runtime and creates a type class.

Recently the midi-type was added. And now we have a module that injects the incoming data from the midi interface into the signal graph. Another module converts the midi-type stream and several number-type outputs. This way, effects with number-types as inputs can be controlled via midi-devices. No changes or rebuilds of the GePhex base system were necessary to add this functionality.

The modules receive typed values and generate others. Why does the engine need to know anything about the types? Isn't it enough that the affected modules know about the type?

It is right that the engine doesn't need many internals of the types to do its job, but there are some actions the engine must take care of:

- Provide some informations like the name to the user
- The renderer must create default values for unconnected inputs
- The value of type objects must be transfered in a serialized form to the user front-end

These the type plugins are like the module plugins shared libraries. They export pointers to functions. The engine then calls these functions if necessary. The symbol names, the signature and the semantic of these are described in the next section.

The c-API

A GePhex data type plugin is a shared library. There is exact one data type in each library file. The file suffix is .so on the Unix platforms and .dll on ms windows system. Each library exports a set of function symbols as defined in the following section.

The GePhex type API consists of a required and an additional part. Every type plugin must implement the required part. The loader of a type plugin must ignore plugins that don't export these symbols. By

implementing functions of the additional part the serialization and automatic subtype conversion functionality can be enabled. But not for all effects these features are necessary.

The first group of function are independent of type instances. The functions `init` and `shutdown` handle the (un)loading of the plugin. `getInfo` and `getSpec` allow the host application to query information about the type from the plugin.

The second group is instance based. There are functions to create and destroy type objects like `newInstance` and `deleteInstance`. Others `assign` and `convertType` instances. The created instances are identified by objects of the type TypeInstanceID this is a unique id with the size of a pointer. It is up to the user of the type plugin to ensure not to mix type object identifier and functions of different types.

There are two optional features a type can provide: (de)serialization and type attributes.

Type attributes describe different representations of values and allow to convert between them. A color is can be in the RGB, YUV or HSV color-space. The color doesn't change if the convert between them. It is just the representation that changes. A color type can have a attribute color-space and some functions to convert transparent from one space to another. Types that have attributes must implement `convertType` and `attributesEqual`.

To store type instances or to transfer them via network it isn't enough to store/transfer the TypeInstanceID we must store the real value not the identifier. The functions `serialize` and `deSerialize` convert type instances to a byte-stream and back.

Required methods

Name

init -- initializes the type plugin

```
int init(void);
void;
```

Description

This function initializes the type plugin. It must be called by the shared library loader after resolving all symbols. It may not be called if any error occures while loading the plugin. No other method of this type class may be called before calling this.

You can e.g. allocate static memory common to all types in this function to use a memory pool.

All resources allocated in init must be deallaocated in the shutDown function.

Return Value

The init function returns a 1 if the type could be initialized and 0 in case of an error. In case of an error no other function (not even the shutdown) may be called.

Name

shutDown -- destructs the type plugin.

```
void shutDown(void);
void;
```

Description

This function closes the type library. Before calling it all type instances must be destructed. It must be called before unloading the dll. After calling shutdown no other method of the type plugin may be called.

You should free all resources (memory, devices) allocated in init here.

Name

getInfo -- querys the user description of the type

```
int getInfo(buf, bufLen);
char* buf;
int bufLen;
```

Arguments

buf
: This is a pointer to a buffer of size bufLen. This buffer can be modified by the method. If the buffer is big enough the info string is written in it.

bufLen
: bufLen is the size of the buffer buf in bytes.

Description

The getInfo function allows the caller to query some information about the type plugin. These are intended for the user presentation and not needed for rendering an effect. At the moment the caller can get a short description of the type. In the future extensions for i18n, icons and color information will be supported.

The semantic of this function is that the caller provides a pointer to an allocated array buf of size bufLen. If the infostring fits in that array the string is written in the buffer and the size of the zero-terminated string is returned. If the buffer is to small no changes to the buffer are applied and the needed size is returned.

The format of the information string is composed like the following example.

```
    "info { name=Palette }"
```

<INFOSTRING> := info { (<ATTRIBUTENAME>=<ATTRIBUTEVALUE>)+ }

At the moment name ist the only attribute that is used.

Return Value

It returns the size of the 0 terminated string written in the buffer. If the buffer was to small to store all requested information the needed size is returned and the buffer stays unchanged.

Name

getSpec -- Returns the spec string of type.

```
const char* getSpec(void);
void;
```

Description

This method returns the specification string of type. This is a c-string with some properties about the type. At the moment this is just the identifier of the type. It is neccessary that this id string is unique cause the type checking in the engine is based on this property.

For a type called IntType the c-string would look like this:

```
"typ_spec { name=typ_IntType; }"
```

<SPECSTRING> := typ_spec { (<ATTRIBUTENAME>=<ATTRIBUTEVALUE>)+ }

The name attribute is the only type property used in this version of the api.

Return Value

The method returns the specification data encoded as a zero terminated string. The pointer to this string is valid till calling shutDown.

Name

newInstance -- creates a new instance of the type

typedef void* TypeInstanceID;

```
TypeInstanceID newInstance();
void;
```

Description

The `newInstance` function creates a new instance of the type. The return value is an identifier with the same memory layout as a pointer. It identifies the typeobject in subsequent calls to the functions of the same type. In most cases this will be a pointer to the memory area allocated to hold the value of the object, but it is not garanteed that this is true. There are different identification mechanisms possible.

The type objects created by this constructor must be destroyed with a call to the same types `deleteInstance` function. Before calling `shutDown` all instances must be deleted.

The created type object has a value, the default value.

Return Value

The function `newInstance` returns a object with the size of a pointer that identifies the type object. This should only passed as an identifier to the type api methods of the same plugin. The caller must ensure not to mix these identifier with the ones of different types.

Name

deleteInstance -- deletes a instance of the type

typedef void* TypeInstanceID;

```
void deleteInstance(instance);
TypeInstanceID instance;
```

Arguments

instance This is the identifier of the instance to delete.

Description

After using a type object for the last time it must be deleted to free all reserved resources like memory or temporary discspace. `instance` identifies the type object created with the `createInstance` function of the same type. The caller must ensure not to mix the type identifiers of one type with the functions of another type. After calling this method the object `instance` is invalid. The instance identifier mustn't used anymore. All type instances created with `newInstance` must be destroyed with `deleteInstance` before calling `shutDown` of their type.

Name

assign -- Assigns the value of the source to the destination typeobject.

typedef void* TypeInstanceID;

```
void assign(destination, source);
TypeInstanceID destination;
TypeInstanceID source;
```

Arguments

destination The value of this type instance will be changed to the value of *source*.

source The value of this type instance is assigned to the *destination* instance. The value of *source* stays unchanged.

Description

`source` and `destination` are identifier for two type objects of the same type as the typeclass. After a call to this function the value of the source object stays unchanged and the value of destination is changed to the one of source. It is the callers task to ensure that both instances and the `assign` function have the same type.

Optional methods

Name

serialize -- serialize the value

typedef void* TypeInstanceID;

```
int serialize(instance, buffer, bufferLen);
TypeInstanceID instance;
char* buffer;
int bufferLen;
```

Arguments

instance
: The value of this type instance will be serialized in the *buffer*. The value of *instance* stays unchanged.

buffer
: This is a pointer to an buffer with the size *bufferLen*. If the buffer is big enough the value of *instance* will be stored serialized in this buffer.

bufferLen
: This is the size of the *buffer* in bytes.

Description

The function serializes the value of *instance* into the *buffer* with the size *bufferLen*. If the buffer provided by the caller isn't big enough for the serialized value the *buffer* is not changed and the required size is returned. In the other case the value of *instance* is written as a bytesequence to the front of the buffer and the number of used bytes is returned.

This method is optional. If you want to provide the (de)serialisation functionality you must also implement the `deSerialize` method.

Return Value

If the buffer is big enough the number of written bytes is returned. In the other case the return value is the number of needed bytes.

Name

deSerialize -- assign the type instance the serilaized value

typedef void* TypeInstanceID;

```
void deSerialize(buffer, bufferLen, instance);
const char* buffer;
int bufferLen;
TypeInstanceID instance;
```

Arguments

instance The value of this type instance will be changed to the one of the *buffer*.

buffer This is a pointer to an buffer of size *bufferLen*. The buffer holds the value that is assigned to the *instance* type object. The buffer will not be changed.

bufferLen This is the size of the *buffer* in bytes.

Description

The function `deSerialize` gets a *buffer* with a serialized value of the type and a instance of the same type. After calling this function the value of type instance is changed to the one in the serialized buffer. The caller must ensure that the value in the *buffer*, the *instance* typeobject and the `deSerialize` function have the same type.

This function is optional. A type plugin that provides (de)serialisation functionality must also implement the inverse function `serialize`.

Name

attributesEqual -- compares attributes with the of one type instance

typedef void* TypeInstanceID; typedef void* TypeAttributesInstanceID;

```
void attributesEqual(instance, attributes);
TypeInstanceID instance;
TypeAttributesInstanceID attributes;
```

Arguments

instance
: The attributes of this type object are compared with the second parameter *attributes*

attributes
: These attributes are compared with the attributes of the *instance*.

Description

The `attributesEqual` function compares the attributes of *instance* with the *attributes*.

An example for an datatype with attributes is the framebuffer type. The resolution is an attribute of the type. There are also other attributes thinkable like colormodel (RGB, BGR, YUV) or the memory layout of the pixels. It must be possible to change the attributes without changing the abstract value of the type instance

This function is optional. Types with attributes must also provide the `convertType` function.

Return Value

The funtion returns 1 if the attributes are equal and 0 in the other cases.

Name

convertType -- Assigns the value from src to dst and changes the attributes while doing that.

typedef void* TypeInstanceID; typedef void* TypeAttributesInstanceID;

```
void convertType(destination, source, attributes);
TypeInstanceID destination;
TypeInstanceID source;
TypeAttributesInstanceID attributes;
```

Arguments

destination
: The value of this type object is changed to the one of *destination* and the attributes of *source* will then equal *attributes*.

source
: The value of this type objects is assigned to *destination* while changing the attributes to *attributes*. The type insance *source* stays unchanged.

attributes
: These attributes are the new attributes of *destination*.

Description

The `changeAttributes` converts the value of the *source* type instance to the *attributes* and assigns the result to the *destination* instance.

This function is optional. Types with attributes must also implement the `attributesEqual` function.

An example for a new data type

The following chapter describes the necessary steps to implement a new data type plugin. The new type will be a color palette. Mathematically this is is a mapping from an interval to the color-space. Since the standard color-space in the GePhex is the red-green-blue color-model with 256 discrete steps from each color-channel and a source space with 256 elements the palette can easily implemented as a array with 256 RGBA entries.

The implementation of the new type is split up in two files: palettetype.h and palettetype.c. The .c file includes the header and will be compiled to the shared library. In the .c files are the exported functions defined. The memory layout of the type and all helper methods resists in the header-file, cause all modules that use the type include the header and do not link with the shared library.

We define the memory layout of the new type in the header in a straight forward way:

```
typedef struct PaletteType_
{
  uint_32 pal[256];
} PaletteType;
```

uint_32 is a typedef for an unsigned integer with 32 bit size. It is defined in the header `basic_types.h`. The 32 bits of the integer are composed by the 4 color components red, green, blue and alpha.

The next step is to define the functions of the shared library. To keep it simple we'll implement only the necessary core methods: `getInfo`,`getSpec`,`deleteInstance`,`newInstance` and `assign`

The implementation of getInfo and getSpec are similarly for all types their propose is to deliver type-specific info strings to the caller. For the new type we set these two strings to:

"typ_spec { name=typ_PaletteType; }" and "info { name=Palette }".

```
const char* getSpec(void)
{
  // return the specification string
  return "typ_spec { name=typ_PaletteType; }";
}

int getInfo (char* buf,int bufLen)
{
  static const char* INFO = "info { name=Palette }";
  int reqLen = strlen(INFO) + 1;
  // check if the buffer is big enough
  if (buf != 0 && reqLen <= bufLen)
    {
      // the string fits in, copy it
      memcpy(buf,INFO,reqLen);
    }
  return reqLen;
```

The other three mandatory functions are the constructor, the destructor and assignment method. In the .c file we place simple wrappers to the real methods in the header. This ensures that modules and the engine use the same implementation since the modules include the type-headers and the engine loads the shared libraries.

```
void* newInstance(void)
{
  return palette_newInstance();
}

void assign(void* dst,const void* src)
{
  palette_assign((PaletteType*)dst,(const PaletteType*)src);
}

void deleteInstance(void* pal)
{
  palette_deleteInstance((PaletteType*) pal);
}
```

The actual implementation of the functions `palette_newInstance`, `palette_assign` and `palette_deleteInstance` is in the header.

The creation of a new type object is split into two functions: one for memory allocation and the other for initialization it with the default value.

```
// initialize a palette with the default value
static __inline void number_initInstance(PaletteType* newType)
{
  int i;
  for(i=0;i!=256;++i)
    {
```

```
      newType->palette[i] = 0x00000000;
    }
}

// allocate memory for a new palette type-object and initialize it
static __inline PaletteType* palette_newInstance(void)
{
  PaletteType* newType = (PaletteType*) malloc(sizeof(PaletteType));
  palette_initInstance(newType);
  return newType;
}
```

The assign method just copies the entries of the source array to the destination.

```
// assign the value of the source palette to the destination palette
static __inline void palette_assign(PaletteType* dst,const PaletteType* src)
{
  dst->palette = src->palette;
int i;
  for(i=0;i!=256;++i)
    {
      dst->palette[i] = src->palette[i];
    }
}
```

The type allocates memory the destructor must free this resource for reuse.

```
/* frees the allocated memory for a palette type object */
static __inline void palette_deleteInstance(PaletteType* pal)
{
  free(pal);
}
```

Adding new effect modules

To add a new effect module, it is best to use pluc, our plugin code generator. It can produce much of the boilerplate code from a simple module specification (see the section called “Pluc the skeleton generator”).

While a module must export plain C functions, it is possible to use other languages for the implementation. Many of GePhex's modules are internally written in C++.

The most important function of a module is update. Here the module can read the input values and produce new output values.

The outputs and needed inputs are always initialized by the engine before the update method is called. The module itself is responsible to convert outputs and inputs if necessary (E.g. some modules want both their input images to have the same resolution, or they want to have the output image to be the same resolution as the input image).

The C-API

A module is a shared library that exports some c functions. In the following section the necessary and

the optional methods and their semantics are described.

The Core Methods

Every module must implement these functions and export their symbol. The loader of the shared library must ignore modules with missing symbols.

Name

init -- initialize the plugin

typedef void (*logT) (int, const char*);

```
int init(logger);
logT logger;
```

Arguments

logger Callback to send log messages from the module plugin to the host application.

Description

The `init` function initialises the module plugin. It is the first function called after loading the shared library. No other method may be called before calling this. It is only allowed to call this function once. The propose is to initialise resources common to all instances of this module class. For example allocation of a memory pool or querys for i/o devices.

If the return value signals an error the caller is not allowed to call another function. The proper handling of such a situation is unloading the shared library.

logger is a pointer to a function that is used to send log messages from the module to the engine. The first parameter of this function is the log level and the second is a const char pointer to the log message as c-string.

Return Value

The return value is 1 if the module could be initialised and 0 in case of an error.

Name

shutDown -- closes the module plugin

```
void shutDown();
void;
```

Description

This function must be called before unloading the shared library. After invoking `shutdown` no other methods of the module may be called. In this method all class-wide resources allocated e.g. by the `init` function must be released. Before calling this method all instances created with `newInstance` must be deleted by calling `deleteInstance`

Name

getSpec -- querys the spefification from the module

```
char* getSpec();
void;
```

Description

This function querys a specification string from the modules plugin. It stores the unique string identifier of the module, the number of inputs and outputs

```
mod_spec
   {
     name = mod_STRING;
     number_of_inputs = UINT;
     number_of_outputs = UINT;
     deterministic = BOOL
   }
```

Return Value

The module specification string is returned as a pointer to a zero terminated char array. This pointer stays valid until calling `shutDown`.

Name

getInfo -- query information for the user interface

```
int getInfo(buffer, bufLen);
char* buffer;
int bufLen;
```

Description

For the dynamic creation of the user interface several information about the module class are necessary:

- icon, name, effect-group
- information about the inputs
- information about the outputs

These information are const that means subsequent calls to this function must return the same value.

The semantic of this function is that the caller gives a pointer to a already allocated array buf of size bufLen and if the info string fits in that array the string is copied and the size of the 0 terminated string is returned. If the buffer is to small no changes to the buffer are applied and the needed size is returned.

Return Value

The function returns the number of written bytes or if the provided buffer was to small the minimum buffer size to store the info string

Name

getInputSpec -- query a description of an input

```
char* getInputSpec(index);
int index;
```

Description

This function returns a pointer to a c-string which describes the input with the *index* number. The format of this specification string looks similar to a structure with default values in programming languages. The structure is called `input_spec` in it has the attributes `type`, `id`, `const`, `default` and `strong_dependency`. "input_spec { type=typ_STRING; id=STRING; const=BOOL; strong_dependency=BOOL; default=STRING}" The order of the attributes is irrelevant. `type` is the unique identifier for the type-class of that input. `id` is an identifier for this input. It has to be unique among all inputs of the module. With the `const` attribute signals the modules to the engine if it wants to change the value of the type object in the `update` function. If the input has set `strong_dependency` to true the engine must always set/update this input before calling `update`. The `default` attribute is the default value of the input when the module is new created. The format to specify this value is the format defined by the `serialise` and `deserialise` functions of the type-class. The specification for some input could look like this: `"input_spec { type=typ_NumberType id=factor const=true strong_dependency=true default=0 }"`

Name

getOutputSpec -- query the specification of an output

```
char* getOutputSpec(index);
int index;
```

Description

Format: "output_spec { type=typ_STRING; id=STRING }"

Name

newInstance -- create a new instance of the module

```
void* newInstance();
void;
```

Description

By calling this method a new instance of the module class is created. The return value is a pointer to the instance. This pointer may only be used by the methods of the same module class. The caller must always ensure that these instance pointers fit to the corresponding functions. There needn't be any internal type check in the module implementation. After using this instance the allocated resources must be released by calling `deleteInstance`.

Name

deleteInstance -- deletes a module instance

```
void deleteInstance(instance);
void* instance;
```

Description

To free the allocated module instance resources the engine calls `deleteInstance`. It must call this destructor after last usage of the module instance. The *`instance`* pointer is invalid after this call and may not be used for further calls.

Name

setInput -- Sets the reference to the typeobject with the input value

```
int setInput(instance, inputIndex, typeObject);
void* instance;
int inputIndex;
void* typeObject;
```

Description

The *typeObject* that holds the value for the input with index *inputIndex* of the module *instance* are set with this function. If the input is declared as const the module may not change the value of the type object because the engine can in this case provide it to another module as input.

Name

setOutput -- sets the referenz to the typeobject to assign the output value

```
int setOutput(instance, outputIndex, typeObject);
void* instance;
int outputIndex;
void* typeObject;
```

Description

A module doesn't create an output type-object with an `update`. It assigns the calculated values to the output objects provided by the caller. This method sets the *`type-object`* for the output with index *`outputIndex`* of the module *`instance`*.

Name

update -- process the inputs and assigns the results to the outputs

```
void update(instance);
void* instance;
```

Description

After all necessary inputs and outputs are set with the `setInput` and `setOutput` the engine can call `update` to assign the results to the outputs.

Name

getInputAttributes -- get type-attributes for input

```
void* getInputAttributes(inputIndex);
int inputIndex;
```

Description

If the type-class of an input supports type attributes an module can force an automatic conversion to fixed attributes. This method returns these attributes. If it returns 0 the input has no fixed attribute or the type doesn't support attributes. inputIndex is the number of the input the caller want to obtain the information.

Optional Methods

To enable additional functionality the shared library must export some of the the following methods. The functions are used for optimization:

- In some cases not all inputs of a module need to be calculated. A switch for example has three inputs and one output. The control input decides which of the two other inputs should be assigned to the output. If the control input is known there is only one of the two inputs needed. We can eliminate the costs for calculating the subtree starting at the unused input by first calculation the control input and when we know its value only calculating one of the two other inputs.
- To copy big type objects like images is expensive. In some cases we know that a module copies the value of an input to an output and makes no or little changes. In this case the engine can eliminate a copy if the input object is exclusively used by only one module.

Name

getPatchLayout -- returns the patch-layout for the next update

```
void getPatchLayout(instance, out2in);
void* instance;
int** out2in;
```

Description

This function returns the patch-layout for the next update. This is a mapping of all outputs to the inputs. If an output is mapped to an input the engine must ensure that before the `update` the output has the same value as the input. If no mapping is requested for an output there are no guarantees for the value of the output object. It must be called direct before `update`. Every output entry holds the index of the input that should be patched to the output. The array has entries for every output. Every output entry holds the index of the input that should be patched to the output. It the entry is -1 the no patching is requested.

Name

strongDependenciesCalculated -- query the needed inputs

```
void strongDependenciesCalculated(instance, neededInputs);
void* instance;
int** neededInputs;
```

Description

This function returns the needed inputs for the next call to `update`. Before calling `strongDependenciesCalculated` the strong-dependency inputs must be updated. The caller provides an array of ints with the same number of elements as inputs. An 0 entry means the value is needed and must be up to date before calling `update`. If the entry for an input is 0 the input needn't be calculated.

Pluc the skeleton generator

The c-api design allows module developers to write their plugins in almost any programming language. This is achieved by a very low-level interface between host and plugin.

Pluc is distributed with GePhex in the `modules` subdirectory.

Many module functions are very simple and have just some lines of code. We want to implement a module that outputs the maximum of its number inputs. In c++ this would be a one-liner:

```
double output1,input1,input2;
output1 = std::max(input1,input2)
```

But the effort needed to export this piece of code via the c-api interface is huge. This is the reason why the pluc.py stub generator exists.

The idea of pluc is that many properties of a module are described in a spec file. Pluc can generate from this file:

- the build-system files (for automake and MS Visual Studio)
- a convenience layer to abstract from the c-api
- a minimum skeleton code for the module implementation that can be used as a basis for implementing the function.

Sample invocations of pluc:

```
pluc.py dsp testmodule.spec
pluc.py am testmodule.spec
pluc.py skel testmodule.spec
```

As can be seen, the first argument to pluc is a command. The second is the filename of the plugin specification.

Table 8.1. Most important pluc commands

am	Creates an automake Makefile.am
dsp	Creates a Visual Studio project
skel	Creates skeleton code for the plugin

Syntax and semantics of the spec file

A spec file consists of three parts. Each part is composed of an identifier followed by a block enclosed in curly braces. The first part contains global settings that determine the behavior of the whole module. The second part contains settings for all inputs of the module. The third part contains settings for all outputs of the module.

Global settings

The global settings begin with a unique name for the module, by convention prefixed with "mod_". To stick with the example from above we choose "mod_max".

Table 8.2. Mandatory global settings

name	STRING
deterministic	BOOL
group	STRING
xpm	FILENAME
author	STRING
version	STRING

name
: the name that is visible to the user

deterministic
: specifies if the module produces the same output whenever the input is the same. This is for example not true for a module that produces random numbers.

group
: allows to group several effects

xpm
: the name of a xpm file that is used as an icon for this module

The following settings are optional:

- enablePatching: BOOL

Input settings

The settings in the input block determine the type, default value and other attributes for every input.

The input block is composed of several blocks, one for each input.

Table 8.3. Mandatory input settings

name	STRING
type	STRING
const	BOOL
strong_dependency	BOOL

name — the name that is visible to the user

type — the type of the input (number, framebuffer, audio, color,...)

const — true iff the module does not change the value of the input

strong_dependency — true iff this input is needed every frame

These attributes are optional.

Table 8.4. Optional input settings

default	STRING
version	STRING

Note

You can provide more optional settings. They are not checked by the engine and can be used to give more information to the gui.

Here is an input block with two number inputs.

```
inputs
{
    lhs
    {
      name              = x
      type              = typ_NumberType
      const             = true
      strong_dependency = true
      default           = 0
      widget_type       = unboundednumber_selector
    }

    rhs
    {
      name              = y
      type              = typ_NumberType
      const             = true
      strong_dependency = true
      widget_type       = unboundednumber_selector
      default           = 0
    }
}
```

Output settings

The settings in the output block determine the type and name for every output.

The output block is composed of several blocks, one for each output.

Table 8.5. Mandatory output settings

name	STRING
type	STRING

Example for an output block with one output:

```
outputs
{
    r
    {
    name                = Result
    type                = typ_NumberType
    }
}
```

An example for a new module

Putting all together, the complete spec for our `max` module looks like this:

```
mod_maxmodule
{
    name                = Maximator
    deterministic       = true
    group               = Number
    xpm                 = maxmodule.xpm
    author              = Zardoz
    version             = 1.0
}

inputs
{
    lhs
    {
      name                = x
      type                = typ_NumberType
      const               = true
      strong_dependency = true
      default             = 0
      widget_type         = unboundednumber_selector
    }

    rhs
    {
      name                = y
      type                = typ_NumberType
      const               = true
      strong_dependency = true
      widget_type         = unboundednumber_selector
      default             = 0
    }
```

```
}

outputs
{
    r
    {
      name              = Result
      type              = typ_NumberType
    }
}
```

Let's assume the spec file is called `maxmodule.spec`. The next step is to produce a `Makefile.am` and the skeleton code:

```
 > pluc.py am maxmodule.spec
 > pluc.py skel maxmodule.spec
```

The skeleton code will be called maxmodule.c. After you have included the `Makefile.am` into your build system, the glue code will be automatically generated when you compile the maxmodule. So whats left to do is to edit the `update` method in `maxmodule.c`:

```
void update(void* instance)
{
  InstancePtr inst    = (InstancePtr) instance;

  inst->out_r->number = max(inst->in_lhs->number, inst->in_rhs->number);
}
```

That's all!