Ceylan's HOWTO

HOW-TO

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Wednesday, November 17, 2021

Lastly updated: Friday, September 2, 2022

Version: 0.0.2

Status: In progress

Dedication: Users of these HOWTOs

Abstract: The role of these `HOW-TOs` is, akin to a cookbook, to share a collection of (technical) recipes ("how-to do this task?) regarding various topics.

These elements are part of the Ceylan umbrella project.

The latest version of this documentation is to be found at the official Ceylan-HOWTOs website (`http://howtos.esperide.org`).

> **Note**
>
> This PDF document includes cross-references between HOWTOs, yet these links make sense only in the context of its HTML counterpart.

Table of Contents

Using the GNU/Linux Operating System

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Sunday, December 19, 2021

Lastly updated: Tuesday, August 30, 2022

Overview

GNU/Linux is our operating system of choice, for many reasons: it is in free software, it is efficient, trustable, reliable and controllable, its mode of operation does not change much over time so any time invested on it is well spent.

Over the years we tried many distributions, including Ubuntu, Debian, Gentoo, Mint.

Our personal all-time favorite is clearly Arch Linux, because it leaves much control to its end user (not attempting to hide details that have to be mastered anyway), it is a "clean" one, driven by a skilled and knowledgeable community, and also because it is a rolling distribution: it updates constantly its packages *without needing to regularly upgrade the whole system*, which would jeopardise it in the same movement (global system updates rarely complete successfully and tend to be postponed because of the many problems they trigger; we found preferable to deal with issues incrementally on a live system - rather than on one that may fail to reboot properly).

It ends up with a very stable, hassle-free distribution, with cutting-edge packages and higher uptimes (several months without needing to reboot), which is desirable for server-like usages.

Software Update

The setup that we use is to perform **automatic nightly updates**. For that we use our update-distro.sh script, run through root's crontab as:

```
$ crontab -l
# Each day at 5:17 AM, update the distro:
17  05   *   *  * /usr/local/bin/update-distro.sh -q
```

As a result, all packages, libraries, executables, etc. are transparently updated, for the best.

However, for a proper management of modules[3], the kernel-related packages shall be special-cased; otherwise after the first kernel update no more modules can be loaded (they will expect to link to that latest installed kernel version, not to the older one being running).

A first line of defense is to force the loading of the modules known to be of interest directly at boot-time, so that they can be for sure loaded and linked to the right kernel.

This may be done by populating `/etc/modules-load.d/` with as many files listing the modules to auto-load, like in:

[3] We tried to rely on DKMS for that, but had still issues with some graphic-related modules, so we preferred managing updates by ourselves.

```
::::::::::::::
/etc/modules-load.d/for-3g-keys.conf
::::::::::::::
option
usb_wwan

::::::::::::::
/etc/modules-load.d/for-all-usb-keys.conf
::::::::::::::
# To be able to mount all kinds of USB keys:
vfat
uas
dm_crypt

::::::::::::::
/etc/modules-load.d/for-mobile-file-transfer.conf
::::::::::::::
# To be able to transfer files between this hosts and mobile phones by MTP:
nls_utf8
isofs
sr_mod
cdrom
# Maybe also: agpgart ahci wdat_wdt wmi_bmof xts

::::::::::::::
/etc/modules-load.d/for-tty-serial-on-usb.conf
::::::::::::::
# To be able to connect tty-like interfaces through a USB port:
ftdi_sio
usbserial

::::::::::::::
/etc/modules-load.d/for-usb-tethering.conf
::::::::::::::
# To enable an Internet access thanks to a smartphone via USB:
usbnet
# Implies:
#rndis_host
#cdc_ether

::::::::::::::
/etc/modules-load.d/for-vlan-support.conf
::::::::::::::
# To be able to manage VLANs:
8021q
```

This is not sufficient though (ex: one cannot anticipate all modules needed after a while); disabling the *automatic* updates of kernels is also key to reduce issues; this can be done by specifying in `/etc/pacman.conf`:

```
IgnorePkg = linux linux-headers
```

LTS (*Long Term Support*) kernels are intentionally *not* listed here, as we prefer having them regularly updated in order to minimise the risk that the base and LTS kernels belong to too close versions (as then a problem in terms of hardware support is more likely to arise at the same time with both).

At least users of NVidia graphic cards may also list there their drivers, as apparently an hardware acceleration supported at boot may be lost after some time, presumably because of an update of its drivers (knowing that the update of the kernel itself was already disabled in that case) - so, if appropriate, better be safe than sorry:

```
IgnorePkg = linux linux-headers nvidia nvidia-utils
```

See also our section about operating system support for 3D.

Updating all packages but kernel-related ones is fine, but of course the latters shall still be also updated appropriately. The best moment for that is just prior to rebooting (knowing that your Linux box never crashes, isn't it?), so for that we use (as root) our shutdown-local-host.sh script, like in:

```
$ shutdown-local-host.sh --reboot
```

The kernel packages, and possibly driver-related ones, will then only be properly updated before the host is rebooted.

Package Management

Configuration

One may enable the multilib repository, which is useful to run 32-bit software on 64-bit hardware. This is useful for example if needing `wine`, knowing that its build from the AUR may fail.

To enable multilib, uncomment in `/etc/pacman.conf`:

```
[multilib]
Include = /etc/pacman.d/mirrorlist
```

then upgrade your system with `pacman -Syu`.

Package-related Commands

- to get information about a package (installed or not): `pacman -Si MY_PACKAGE`
- to list all packages explicitly installed and not required as dependencies: `pacman -Qet`
- to determine which package installed a specified file:
  - on Arch: `pkgfile SOME_FILE`; `pkgfile` itself must have been installed beforehand, with `pacman -S pkgfile`, and be updated, with `pkgfile --update` (still as root)
  - on Debian: `apt-file search SOME_FILE` after a similar initial install thanks to: `sudo apt-get install apt-file && sudo apt-file update`

  - for many distros, one may rely on the command-not-found website

- to determine which package owns (would install) specified file(s): `pacman -Qo FILES`

See this page for many more Arch-related commands.

Interesting Packages

They might be lesser known:

- `cpulimit`: the way of limiting CPU usage of a given process, for example to avoid overheat (`nice` just defines respective process priorities)
- `inotify-tools`: to be able to monitor filesystem events (ex: with `inotifywait`) from scripts
- `jq`: for command-line JSON processing (ex: `jq . myfile.json` to display it properly on a terminal)
- `mathjax`: to generate LaTeX-like images for the web
- `most`: a replacement for `more`
- `pdftk`: to transform PDF files
- `pkgfile`: to retrieve file information about packages

Systemd-related Hints

Systemd is the current reference system and service manager for GNU/Linux. It is in charge of configuring, launching, monitoring, controlling, etc. all the software services running on a given host.

Systemd Commands

- **listing** the units:
  - managed by systemd: `systemctl list-units [PATTERN]`
  - installed (as files): `systemctl list-unit-files [PATTERN]` or `tree /etc/systemd/system`
- **getting runtime status information** about units: `systemctl status [PATTERN]`
- **controlling** units: `systemctl start|stop|restart [PATTERN]`
- **reloading**:
  - a service-specific configuration of a unit: `systemctl reload [PATTERN]` (ex: requesting Apache to reload its own httpd.conf file)
  - the systemd configuration file of a unit: `systemctl daemon-reload [PATTERN]` (ex: reloading the `apache.service` systemd unit file)
- **enabling/disabling** for good units (while not starting/stopping them): `systemctl enable|disable [PATTERN]`

Systemd Journal

In order to query the contents of the systemd journal (as written by `systemd-journald.service`), the `journalctl` command may be used.

To consult the journal:

- from:
  - the oldest entry collected: `journalctl`
  - last boot: `journalctl -b`
  - the most recent journal entries, listing them to current end, and:
    - stopping there: `journalctl -e`
    - continuously printing the new entries as they are added: `journalctl -f` (or `--follow`)
- for a given unit `my_unit` (see Systemd Commands for selection): `journalctl -u my_unit`
- showing `COUNT` lines: add `-n COUNT` (default: 10)

Process-related Post-Mortem Investigations

Sometimes a UNIX process crashes and, typically if one developed it, one wants to investigate the issue, based on a core dump produced by the operating system.

This Arch Linux article will give all relevant details.

In short, `coredumpctl list` will list all known core dumps from oldest to most recent, such as in:

```
$ coredumpctl list
TIME                            PID   UID  GID SIG     COREFILE EXE                SIZE
[...]
Tue 2021-12-21 20:53:02 CET 73873 1007 988 SIGSEGV present  [...]/bin/beam.smp 14.6M
```

The last core dump produced may be studied directly, thanks to `coredumpctl debug`, relying on `gdb` to fetch much lower-level information:

```
$ coredumpctl debug
           PID: 73873 (beam.smp)
           UID: 1007 (xxx)
           GID: 988 (users)
        Signal: 11 (SEGV)
     Timestamp: Tue 2021-12-21 20:53:01 CET (38min ago)
  Command Line: /home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/erts-12.2/bin/beam.smp
    Executable: /home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/erts-12.2/bin/beam.smp
 Control Group: /user.slice/user-1007.slice/session-2.scope
          Unit: session-2.scope
         Slice: user-1007.slice
       Session: 2
     Owner UID: 1007 (xxx)
       Boot ID: f8abe9473f7e4fea8ba24944e35ce7d9
    Machine ID: c9413a71e7b4498f831e2df7a08e5f33
```

```
     Hostname: xxx
      Storage: /var/lib/systemd/coredump/core.beam\x2esmp.1007.f8abe9473f7e4fea8ba2494
    Disk Size: 14.6M
      Message: Process 73873 (beam.smp) of user 1007 dumped core.

               Found module /home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/erts-12.2
               Found module /home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/lib/wx-2
[...]
   Stack trace of thread 74039:
   #0  0x00007f6e5461a74b __memmove_avx_unaligned_erms (libc.so.6 + 0x16374b)
   #1  0x00007f6d8a204428 n/a (iris_dri.so + 0xd12428)
   #2  0x00007f6d89733207 n/a (iris_dri.so + 0x241207)
   #3  0x00007f6d89733c97 n/a (iris_dri.so + 0x241c97)
   #4  0x00007f6d898d8b0d n/a (iris_dri.so + 0x3e6b0d)
   #5  0x00007f6d898d8bf2 n/a (iris_dri.so + 0x3e6bf2)
   #6  0x00007f6d8b2f241c n/a (/home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/lib/wx
[New LWP 74039]
[New LWP 73873]
[...]
Core was generated by `/home/xxx/Software/Erlang/Erlang-24.2/lib/erlang/erts-12.2/bin/
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007f6e5461a74b in __memmove_avx_unaligned_erms () from /usr/lib/libc.so.6
[Current thread is 1 (Thread 0x7f6d900aa640 (LWP 74039))]
```

Then:

```
(gdb) bt
[...]
#0  0x00007f6e5461a74b in __memmove_avx_unaligned_erms () at /usr/lib/libc.so.6
#1  0x00007f6d8a204428 in  () at /usr/lib/dri/iris_dri.so
#2  0x00007f6d89733207 in  () at /usr/lib/dri/iris_dri.so
#3  0x00007f6d89733c97 in  () at /usr/lib/dri/iris_dri.so
#4  0x00007f6d898d8b0d in  () at /usr/lib/dri/iris_dri.so
#5  0x00007f6d898d8bf2 in  () at /usr/lib/dri/iris_dri.so
#6  0x00007f6d8b2f241c in ecb_glTexImage2D(ErlNifEnv*, ErlNifPid*, ERL_NIF_TERM*) (env
[...]
#29 0x00007f6d92967188 in wxe_main_loop(void*) (_unused=<optimized out>) at wxe_main.c
```

(this example was an Erlang wx/OpenGL-oriented crash)

From there, standard gdb-fu shall be sufficient to give much insight. Once done, use `q` to quit.

Filesystem-related Issues

Word of wisdom: perform backups. Regularly. Safely stored.

Target is busy

Umounting may fail with `umount: /var/foobar: target is busy`. Finding the culprit can be done with:

```
$ lsof /var/foobar
```

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
bash 275648 root cwd DIR 254,1 20480 45236687 /var/foobar/www/Foo

Failed I/O Operations

Some operations (even a `touch` done by `root`) may fail, reporting `Read-only file system`; `systemd-fsck` may also report `Structure needs cleaning`.

Check with `mount` that a given filesystem/partition is (still) in the expected state:

```
# For example:
/dev/mapper/foobar on /var/foobar type ext4 (rw,relatime,data=ordered)

# may have become:
/dev/mapper/foobar on /var/foobar type ext4 (ro,relatime,errors=continue,data=ordered)
```

Best option here is to unmount, make a sector-level backup of the disk, and then check it. Probably that the disk already failed and you will have to resort to revert to a former backup.

Checking a Disk

This can be done with `e2fsck` once the filesystem is unmounted, like in: `e2fsck -cDp /dev/mapper/foobar`, with:

- `-c`: use `badblocks` to do a read-only scan of the device
- `-D`: optimize directories in the filesystem
- `-p`: automatically repair ("preen") the file system

If having errors like:

```
e2fsck: Input/output error while trying to open /dev/mapper/foobar‘‘
```

or:

```
The superblock could not be read or does not describe a valid
ext2/ext3/ext4 filesystem [...]
```

then, unless you are trying to check an encrypted partition (ex: not having cryptsetup-opened a LUKS one) or a superblock copy is enough, prospects are grim.

Try for example `fdisk` / `gdisk` / `cgdisk`, `parted`, `smartctl --all -T verypermissive`, `badblocks -o ~/scan-result.txt -sv` (for which errors are counted as NumberOfReadErrors / NumberOfWriteErrors / NumberOfCorruptionErrors) - yet your disk is maybe already unrecoverable.

Quick Topics

Installing Wine

Install it, once enabling multilib has been done, with: `pacman -S wine`.

When run, this may lead `wine-mono` to be auto-installed.

The pseudo-Windows filesystem is then located mostly in `~/.wine/drive_c`.

Invalid PGP Signatures in Packages

This happens regularly, symptom being: `File /var/cache/pacman/pkg/xxx.pkg.tar.zst is corrupted (invalid or corrupted package (PGP signature))`.

Solution: `pacman -S archlinux-keyring` is at least often enough. Otherwise `pacman-key --populate archlinux`, or changing mirror might help.

Adding a Locale

On some hosts, issues in terms of a lacking locales may be reported, like in the following:

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
```

This can be fixed by uncommenting the corresponding locale (`en_US.UTF-8` here) in `/etc/locale.gen`, and then regenerating the system locales by running (as root) `locale-gen`:

```
$ locale-gen
Generating locales (this might take a while)...
 en_US.UTF-8... done
 fr_FR.UTF-8... done
 fr_FR.ISO-8859-15@euro... done
Generation complete.
```

Applying a XFCE4 configuration to a different user

The objective is to duplicate the configuration of the `user_a` desktop to the `user_b` one, for example if `user_b` is used to test an untrusted application.

A key point is to ensure that at least `user_b` has no ongoing graphical session (otherwise any new configuration may be overwritten with an older one at logout).

Then, typically as root:

```
$ rm -rf /home/user_b/.config/{xfce4,Thunar} /home/user_b/.local/share/xfce4
$ cp -rf /home/user_a/.config/{xfce4,Thunar} /home/user_b/.config
$ cp -rf /home/user_a/.local/share/xfce4 /home/user_b/.local/share
$ chmod -R /home/user_b/{.config,.local} user_b:users
```

Then a key point is to fully reset XFCE4. Unlogging and/or `killall -HUP xfdesktop` may not be sufficient. As a last resort, rebooting shall work.

Selecting a per-user wallpaper may be of help.

Emacs

Configuration Our init.el, that may be stored in `~/.emacs.d/`, currently relies on straight.el.

Hints

- to tune the font size:
  - temporarily: use `C-x C-+` to increase, `C-x C--` to decrease (or press Shift then click the first mouse button to select a relevant option)
  - permanently: one may add in one's Emacs configuration file for example: `(set-face-attribute 'default nil :height 100)`
- to insert in the current buffer the output of a shell command: `Ctrl-U then Alt-!`: enter that shell command, whose output will be pasted at the current cursor position
- to prefix all lines of the selected region: `Ctr-X then r then t`, then type the prefix then type Enter
- to sort alphabetically the selected region: `Meta-x sort-lines`

Other useful commands to trigger with `Meta-x`:

- `replace-string`
- `query-replace`
- `kill-rectangle`
- `Ctrl-g` to abort the current entry in the minibuffer

Troubleshooting Sometimes problems arise due to older packages, this may be solved by removing the `~/.emacs.d/straight` directory, relaunching Emacs and waiting for all related clones to be downloaded again.

Other Settings

See our preferences for Startpage; it can be set as homepage.

Shortcuts

Listing here the main keyboard shortcuts of interest for a few base tools.

For `less` Mix of `vi` / `more` conventions, in order to:

- **forward-search** for the N-th line containing the pattern (N defaults to 1): `/pattern`; use `?/pattern` to **backward-search**
- **repeat previous search**: `n`:
- **invoke an editor** on the current file being viewed: `v`
- **go to next file**: `:n`; for the **previous** one: `:p`
- **help**: `h`
- **quit**: `q`

For `mplayer`

- **pause/unpause** the current playback: `<space>`
- **decrease the volume**: `/`; to **increase** it: `*`
- **go backward/forward** in the current playback: `left` and `right arrow` keys
- **jump to next playback**: `<Enter>` or `<Escape>`
- **stop** all playbacks: `<CTRL-C>`
- **display durations**: `o` to toggle OSD (*On-Screen Display*), i.e. elapsed, elapsed / total, etc.
- **take a screenshot**: `s` (notably if using our v script, as the `-vf screenshot` command-line option must have been specified); then a `shot0001.png` file will be silently created in the directory whence mplayer was launched, next screenshot will be `shot0002.png`, etc.

See Also

One may refer to our other mini-HOWTO regarding:

- Network Management
- Cybersecurity

The Ceylan-Hull section system-related section might also be of interest.

Erlang

Organisation: Copyright (C) 2021-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Tuesday, August 30, 2022

Overview

Erlang is a concurrent, functional programming language available as free software; see its official website for more details.

Erlang is dynamically typed, and is executed by the BEAM virtual machine. This VM (*Virtual Machine*) operates on bytecodes and can perform Just-In-Time compilation. It powers also other related languages, such as Elixir and LFE.

Let's Start with some Shameless Advertisement for Erlang and the BEAM VM

Taken from this presentation:

> **Hint**
>
> *What makes Elixir StackOverflow's #4 most-loved language?*
> *What makes Erlang and Elixir StackOverflow's #3 and #4 best-paid languages?*
> *How did WhatsApp scale to billions of users with just dozens of Erlang engineers?*
> *What's so special about Erlang that it powers CouchDB and RabbitMQ?*
> *Why are multi-billion-dollar corporations like Bet365 and Klarna built on Erlang?*
> *Why do PepsiCo, Cars.com, Change.org, Boston's MBTA, and Discord all rely on Elixir?*
> *Why was Elixir chosen to power a bank?*
> *Why does Cisco ship 2 million Erlang devices each year? Why is Erlang used to control 90% of Internet traffic?*

Installation

Erlang can be installed thanks to the various options listed in these guidelines.

Building Erlang from the sources of its latest stable version is certainly the best approach; for more control we prefer relying on our custom procedure.

For a development activity, we recommend also specifying the following options to our `conf/install-erlang.sh` script:

- `--doc-install`, so that the reference documentation can be accessed locally (in `~/Software/Erlang/Erlang-current-documentation/`); creating a bookmark pointing to the module index, located in `doc/man_index.html`, would most probably be useful

- `--generate-plt` in order to generate a PLT file allowing the static type checking that applies to this installation (may be a bit long and processing-intensive, yet it is to be done once per built Erlang version)

Run `./install-erlang.sh --help` for more information.

Once installed, ensure that `~/Software/Erlang/Erlang-current-install/bin/` is in your PATH (ex: by enriching your `~/.bashrc` accordingly), so that you can run `erl` (the Erlang interpreter) from any location, resulting a prompt like:

```
$ erl
Erlang/OTP 24 [erts-12.1.5] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]

Eshell V12.1.5  (abort with ^G)
1>
```

Then enter `CTRL-C` twice in order to come back to the (UNIX) shell.

Congratulations, you have a functional Erlang now!

To check from the command-line the version of an Erlang install:

```
$ erl -eval '{ok, V} = file:read_file( filename:join([code:root_dir(), "releases", er
24.2
```

Ceylan's Language Use

Ceylan users shall note that most of our related developments (namely Myriad, WOOPER, Traces, LEEC, Seaplus, Mobile, US-Common, US-Web and US-Main) depart significantly from the general conventions observed by most Erlang applications:

- notably because of their reliance on parse transforms, by default they rely on our own build system based on GNU make (rather than on rebar3)
- they tend not to rely on OTP abstractions such as `gen_server`, as WOOPER offers OOP (*Object-Oriented Programming*) ones that we prefer

Using the Shell

If it is as simple to run `erl`, we prefer, with Ceylan settings, running `make shell` in order to benefit from a well-initialized VM (notably with the full code path of the current layer and the ones below).

Refer then to the shell commands, notably for:

- `f/1`, used as `f(X).` in order to *forget* a variable `X`, i.e. to remove the binding of this variable and be able to (re)assign it afterwards
- `l/1` (apparently undocumented), used as `l(my_module)`, to (re)load that module, purging any old version of it; convenient to reload also specially-built BEAMs (ex: based on parse transforms) that may have been compiled behind the scene
- `rl/1` is *not* "reload (module)" (use `l/1` for that), it is "record list" (it prints selected record definitions)

- `c/1`, used as `c(my_module)`, to compile (if necessary) *and* (re)load that module, purging any old version of it
- `rr/{1,2,3}` (ex: used as `rr(Path).`) to *read records* and have them available on the shell; for example, to be able to use the records defined by xmerl:

```
1> rr(code:lib_dir(xmerl) ++ "/include/xmerl.hrl").
```

See also the JCL mode (for *Job Control Language*) to connect and interact with other Erlang nodes.

About Security

- one should not encrypt messages directly with a key pair (ex: with RSA, only messages up to around 200 bytes long can be encrypted): one should encrypt only a *symmetric key* (generated by a cryptographically-safe random algorithm) that is then used to encrypt one's message(s); ensure an Encrypt-Then-Authenticate scheme to prevent padding oracle attacks (and a secure-compare algorithm for the *Message Authentication Code* verification to prevent timing attacks); using libsodium should make mistakes using the standard crypto primitives less error-prone; see the enacl Erlang binding for that; for more information, refer to the corresponding thread
- relevant sources of information:
  - books:
    - `Cryptography Engineering: Design Principles and Practical Applications`
    - `Practical Cryptography`
  - the Security Working Group of the EEF (*Erlang Ecosystem Foundation*)

OTP Guidelines

The `.app` Specification

For an overall application named `Foobar`, we recommend defining in `conf/foobar.app.src` an application specification template that, once properly filled regarding the version of that application and the modules that it comprises (possibly automatically done thanks to the Ceylan-Myriad logic), will result in an actual application specification file, `foobar.app`.

Such a file is necessary in all cases, to generate an OTP application (otherwise with rebar3 nothing will be built), an OTP release (otherwise the application dependencies will not be reachable), and probably an hex package as well.

This specification content is to end up in various places:

- in `ebin/foobar.app`
- if using rebar3, in the OTP build tree (by default: `./_build/lib/foobar/ebin/foobar.app`)

- with `src/foobar.app.src` being a symbolic link pointing to `ebin/foobar.app` (probably at least for hex)

Starting OTP Applications

For an OTP *active* application of interest - that is one that provides an actual service, i.e. running processes, as opposed to a mere *library* application, which provides only code - such a specification defines, among other elements, which module will be used to start this application. We recommend to name this module according to the target application and to suffix it with `_app`, like in:

```
{application, foobar, [
    [...]
    {mod, {foobar_app, [hello]}},
    [...]
```

This implies that once a user code will call `application:start(foobar)`, then `foobar_app:start(_Type=normal, _Args=[hello])` will be called in turn.

This `start/2` function, together with its `stop/1` reciprocal, are the functions listed by the OTP (active) `application` behaviour; at least for clarity, it is better that `foobar_app.erl` comprises `-behaviour(application).`

Pre-Launch Functions

The previous OTP callbacks may be called by specific-purpose launching code; we tend to define an `exec/0` function for that: then, with the Myriad make system, executing on the command-line `make foobar_exec` results in `foobar_app:exec/0` to be called.

Having such a pre-launch function is useful when having to set specific information beforehand (see `application:set_env/{1,2}`) and/or when starting by oneself applications (ex: see `otp_utils:start_applications/2`).

In any case this should result in `foobar_app:start/2` to be called at application startup, a function whose purpose is generally to spawn the root supervisor of this application.

OTP Supervisors

The purpose of supervisors is to ease the development of fault-tolerant applications by building hierarchical process structures called *supervision trees*.

For that, supervisors are to monitor their children, that may be workers (typically implementing the `gen_{event,server,statem}` behaviour) and/or other supervisors (they can thus be nested).

We recommend to define a `foobar_sup:start_link/0` function (it is an user-level API, so any name and arity can be used). This `foobar_sup` module is meant to implement the `supervisor` behaviour (to be declared with `-behaviour(supervisor).`), which in practice requires an `init/1` function to be defined.

So this results, in `foobar_sup`, in a code akin to:

```
-spec start_link() -> supervisor:startlink_ret().
start_link() ->
```

```
    % This will result in calling init/1 next:
    supervisor:start_link( _Registration={local, my_foobar_main_sup},
                           _Mod=?MODULE, _Args=[]).

-spec init(list()) -> {'ok', {supervisor:sup_flags(), [child_spec()]}}.
init(_Args=[]) ->
    [...]
    {ok, {SupSettings, ChildSpecs}}.
```

Declaring Worker Processes

Our `otp_utils` module may help a bit defining proper restart strategies and child specifications, i.e. the information regarding the workers that will be supervised, here, by this root supervisor.

For example it could be:

```
init(_Args=[]) ->
    [...]
    SupSettings = otp_utils:get_supervisor_settings(
                    _RestartStrategy=one_for_one, ExecTarget),
    % Always restarted in production:
    RestartSettings = otp_utils:get_restart_setting(ExecTarget),
    WorkerShutdownDuration =
        otp_utils:get_maximum_shutdown_duration(ExecTarget),
    % First child, the main Foobar worker process:
    MainWorkerChild = #{
        id => foobar_main_worker_id,
        start => {_Mod=foobar, _Fun=start_link,
                  _MainWorkerArgs=[A, B, C]},
        restart => RestartSettings,
        shutdown => WorkerShutdownDuration,
        type => worker,
        modules => [foobar] },
    ChildSpecs = [MainWorkerChild],
    {ok, {SupSettings, ChildSpecs}}.
```

Children are created synchronously and in the order of their specification.

So if `ChildSpecs=[A, B, C]`, then a child according to the A spec is first created, then, once it is over (either its `init/1` finished successfully, or it called `proc_lib:init_ack/{1,2}`[4] and then continued its own initialisation concurrently), a child according to the B spec is created, then once done a child according to the C spec.

Implementing Worker Processes

Such a worker, which can be any Erlang process (implementing an OTP behaviour, like `gen_server`, or not) will thus be spawned here through a call to the `foobar:start_link/3` function (another user-defined API) made by this

[4] Typically: `proc_lib:init_ack(self())`.

supervisor. This is a mere *call* (an `apply/3`), not a spawn of a child process based on that function.

Therefore the called function is expected to *create* the worker process by itself, like, in the `foobar` module:

```
start_link(A, B ,C) ->
   [...]
   {ok, proc_lib:start_link(?MODULE, _Func=init,
       _Args=[U, V], _Timeout=infinity, SpawnOpts)}.
```

Here thus the spawned worker will start by executing `foobar:init/2`, a function not expected to return, often trapping EXIT signals (`process_flag(trap_exit, true)`), setting system flags and, once properly initialised, notifying its supervisor that it is up and running (ex: `proc_lib:init_ack(_Return=self())`) before usually entering a tail-recursive loop.

Terminating Workers

Depending on the `shutdown` entry of its child specification, on application stop that worker may be terminated by different ways. We tend to prefer specifying a maximum shutdown duration: then the worker will be sent by its supervisor first a `shutdown` EXIT message, that this worker may handle, typically in its main loop:

```
receive
   [...]

   {'EXIT', _SupervisorPid, shutdown} ->
      % Just stop.
      [...]
```

If the worker fails to stop (on time, or at all) and properly terminate, it will then be brutally killed by its supervisor.

Supervisor Bridges

Non-OTP processes (ex: WOOPER instances) can act as supervisors thanks to the supervisor_bridge module.

Such a process shall implement the `supervisor_bridge` behaviour, namely `init/1` and `terminate/2`. If the former function spawns a process, the latter shall ensure that this process terminates in a synchronous manner, otherwise race conditions may happen.

See traces_bridge_sup for an example thereof.

Extra Information

One may refer to;

- Ceylan-Myriad as an example of OTP *Library* Application
- Ceylan-WOOPER or Ceylan-Traces as examples of OTP *Active* Applications
- US-Web as an example of most of these supervisor principles

More Advanced Topics

Metaprogramming

Metaprogramming is to be done in Erlang through **parse transforms**, which are user-defined modules that transform an AST (for *Abstract Syntax Trees*, an Erlang term that represents actual code; see the Abstract Format for more details) into another AST that is fed afterwards to the compiler.

See also:

- this introduction to parse transforms
- Ceylan-Myriad's support for metaprogramming

Improper Lists

A proper list is created from the empty one (`[]`, also known as "nil") by appending (with the `|` operator, a.k.a. "cons") elements in turn; for example `[1,2]` is actually `[1 | [2 | []]]`.

However, instead of enriching a list from the empty one, one *can* start a list with any other term than `[]`, for example `my_atom`. Then, instead of `[2|[]]`, `[2|my_atom]` may be specified and will be indeed a list - albeit an improper one.

Many recursive functions expect proper lists, and will fail (typically with a function clause) if given an improper list to process (ex: `lists:flatten/1`).

So, why not banning such construct? Why even standard modules like `digraph` rely on improper lists?

The reason is that improper lists are a way to reduce the memory footprint of some datastructures, by storing a value of interest instead of the empty list.

Indeed, as explained in this post, a (proper) list of 2 elements will consume:

- 1 list cell (2 words of memory) to store the first element and a pointer to second cell
- 1 list cell (2 more words) to store the second element and the empty list

For a total of 4 words of memory (so, on a 64-bit architecture, it is 32 bytes).

As for an improper list of 2 elements, only 1 list cell (2 words of memory) will be consumed to store the first element and then the second one.

Such a solution is even more compact than a pair (a 2-element tuple), which consumes 2+1 = 3 words. Accessing the elements of an improper list is also faster (one handle to be inspected vs also an header to be inspected).

Finally, for sizes expressed in bytes:

```
1> system_utils:get_size([2,my_atom]).
40

2> system_utils:get_size({2,my_atom}).
32

3> system_utils:get_size([2|my_atom]).
24
```

See also the 1, 2 pointers for more information.

Everyone shall decide on whether relying on improper lists is a trick, a hack or a technique to prohibit.

OpenCL

Open Computing Language is a standard interface designed to program many processing architectures, such as CPUs, GPUs, DSPs, FPGAs.

OpenCL is notably a way of using one's GPU to perform more general-purpose processing than typically the rendering operations allowed by GLSL (even compared to its compute shaders).

In Erlang, the cl binding is available for that.

A notable user thereof is Wing3D; one may refer to the `*.cl` files in this directory, but also to its optional build integration as a source of inspiration, and to wings_cl.erl.

Post-Mortem Investigations

Erlang programs may fail, and this may result in mere (Erlang-level) crashes (the VM detects an error, and reports information about it, possibly in the form of a crash dump) or (sometimes, quite infrequently though) in more brutal, lower-level core dumps (the VM crashes as a whole, like any faulty program run by the operating system); this last case happens typically when relying on faulty NIFs.

Erlang Crash Dumps If experiencing "only" an Erlang-level crash, a `erl_crash.dump` file is produced in the directory whence the executable (generally `erl`) was launched. The best way to study it is to use the `cdv` (refer to crashdump viewer) tool, available, from the Erlang installation, as `lib/erlang/cdv`[5].

Using this debug tool is as easy as:

```
$ cdv erl_crash.dump
```

Then, through the wx-based interface, a rather large number of Erlang-level information will be available (processes, ports, ETS tables, nodes, modules, memory, etc.) to better understand the context of this crash and hopefully diagnose its root cause.

Core Dumps In the worst cases, the VM will crash like any other OS-level process, and generic (non Erlang-specific) tools will have to be used. Do not expect to be pointed to line numbers in Erlang source files anymore!

Refer to our general section dedicated to core dumps for that.

Language Bindings

The two main approaches in order to integrate third-party code to Erlang are to:

[5] Hence, according to the Ceylan-Myriad conventions, in `~/Software/Erlang/Erlang-current-install/lib/erlang/cdv`.

- interact with it as if it was another Erlang node; we defined Ceylan-Seaplus for that purpose
- directly link the current Erlang VM to this code, through NIF; it can be done manually, or may be automatised thanks to nifty; this can be especially useful for larger APIs (ex: SDL)

Language Implementation

Message-Passing: Copying vs Sharing

Knowing that, in functional languages such as Erlang, terms ("variables") are immutable, why could not they be shared between local processes when sent through messages, instead of being copied in the heap of each of them, as it is actually the case with the Erlang VM?

The reason lies in the fact that, beyond the constness of these terms, their life-cycle has also to be managed. If they are copied, each process can very easily perform its (concurrent, autonomous) garbage collections. On the contrary, if terms were shared, then reference counting would be needed to deallocate them properly (neither too soon nor never at all), which, in a concurrent context, is bound to require locks.

So a trade-off between memory (due to data duplication) and processing (due to lock contention) has to be found and at least for most terms (excepted larger binaries), the sweet spot consists in sacrificing a bit of memory in favour of a lesser CPU load. Solutions like persistent_term may address situations where more specific needs arise.

Just-in-Time Compilation

This long-awaited feature, named *BeamAsm* and whose rationale and history have been detailed in these articles, has been introduced in Erlang 24 and shall transparently lead to increased performances for most applications.

Static Typing

Static type checking can be performed on Erlang code; the usual course of action is to use Dialyzer - albeit other solutions like Gradualizer and eqWAlizer exist.

A few statically-typed languages can operate on top of the Erlang VM, even if none has reached yet the popularity of Erlang or Elixir (that are dynamically-typed).

In addition to the increased type safety that statically-typed languages permit (possibly applying to sequential code but also to inter-process messages), it is unsure whether such extra static awareness may also lead to better performances (especially now that the standard compiler supports JIT).

Intermediate Languages

To better discover the inner workings of the Erlang compilation, one may look at the eplaypen online demo (whose project is here) and/or at the Compiler Explorer (which supports the Erlang language among others).

Both of them allow to read the intermediate representations involved when compiling Erlang code (BEAM stage, erl_scan, preprocessed sources, abstract

code, Core Erlang, Static Single Assignment form, BEAM VM assembler opcodes, x86-64 assembler generated by the JIT, etc.).

Short Hints

Dealing with Conditional Availability

Regarding modules, and from the command-line Depending on how Erlang was built in a given environment, some modules may or may not be available.

A way of determining availability and/or version of a module (ex: `wx`, `cl`, `crypto`) from the command-line:

```
$ erl -noshell -eval 'erlang:display(code:which(wx))' -s erlang halt
"/home/bond/Software/Erlang/Erlang-24.2/lib/erlang/lib/wx-2.1.1/ebin/wx.beam"

$ erl -noshell -eval 'erlang:display(code:which(cl))' -s erlang halt
non_existing
```

A corresponding in-makefile test, taken from Wings3D:

```
# Check if OpenCL package is as external dependency:
CL_PATH = $(shell $(ERL) -noshell -eval 'erlang:display(code:which(cl))' -s erlang hal
ifneq (,$(findstring non_existing, $(CL_PATH)))
   # Add it if not found:
   DEPS=cl
endif
```

Regarding language features, from code Some features appeared in later Erlang versions, and may be conditionally enabled.

For example:

```
FullSpawnOpts = case erlang:system_info(version) >= "8.1" of

   true ->
       [{message_queue_data, off_heap}|BaseSpawnOpts];

   false ->
       BaseSpawnOpts

end,
[...]
```

Runtime Library Version

To know **which version of a library** (here, wxWidgets) a given Erlang install is using (if any), one may run an Erlang shell (`erl`), collect the PID of this (UNIX) process (ex: `ps -edf | grep beam`), then trigger a use of that library (ex: for `wx`, execute `wx:demo().`) in order to force its dynamic binding.

Then determine its name, for example thanks to `pmap ${BEAM_PID} | grep libwx`).

This may indicate that for example `libwx_gtk2u_core-3.0.so.0.5.0` is actually used.

Proper Naming

Variable Shorthands Usually we apply the following conventions:

- the *head* and *tail* of a list are designated as `H` and `T`, like in: `L = [H|T]`
- `Acc` means *accumulator*, in a tail-recursive function
- `K` designates a *key*, and `V` designates its associated *value*
- a list of elements is designated by a plural variable name, usually suffixed with `s`, like in: `Ints`, `Xs`, `Cars`

Function Pairs To better denote reciprocal operations, following namings for functions may be used:

- for services:
  - activation: `start` / `stop`
  - setup: `init` / `terminate`
- for instances:
  - life-cycle: `new` / `delete`
  - construction/destruction: `create` / `destruct` (ex: avoid `destroy` there)

Using wx

wx is now[6] the standard Erlang way of programming one's graphical user interface; it is a binding to the wxWidgets library.

Here are some very general wx-related information that may be of help when programming GUIs with this backend:

- in wxWidgets parlance, "window" designates any kind of widget (not only frame-like elements)
- if receiving errors about `{badarg,"This"}`, like in:

```
{'_wxe_error_',710,{wxDC,setPen,2},{badarg,"This"}}
```

[6] `wx` replaced gs. To shelter already-developed graphical applications from any future change of backend, we developed MyriadGUI, an interface doing its best to hide the underlying graphical backend at hand: should the best option in Erlang change, that API would have to be ported to the newer backend, hopefully with little to (ideally) no change in the user applications.

it is probably the sign that the user code attempted to perform an operation on an already-deallocated wx object; the corresponding life-cycle management might be error-prone, as some deallocations are implicit, others are explicit, and in a concurrent context race conditions easily happen

- if creating, from a wx-using process, another one, this one should set a relevant environment first (see `wx:set_env/1`) before using wx functions
- the way wx/wxWidgets manage event propagation is complex; here are some elements:
  - for each actual event happening, wx creates an instance of wxEvent (a direct, abstract child class of `wxObject`), which itself is specialised into many event classes that are more concrete
  - among them, they are so-called *command events*, which originate from a variety of simple controls and correspond to the wxCommandEvent mother class; by default only these command events are set to propagate, i.e. only them will be transmitted to the event handler of the parent widget if the current one does not process them by itself ("did not connect to them")
  - by default, for a given type of event, when defining one's event handler (typically when connecting a process to the emitter of such events), this event will *not* be propagated anymore, possibly preventing the built-in wx event handlers to operate (ex: to properly manage resizings); to restore their activation, `skip` (to be understood here as "propagate event" - however counter-intuitive it may seem) shall be set to `true` (either by passing a corresponding option when connecting, or by calling `wxEvent:skip/2` with `skip` set to `true` from one's event handler)[7]
  - when a process connects to the emitter of a given type of events (ex: `close_window` for a given frame), this process is to receive corresponding messages and then perform any kind of operation; however these operations cannot be synchronous (they are non-blocking: the process does not send to anyone any notification that it finished handling that event), and thus, if `skip` is `true` (that is, if event propagation is enabled), any other associated event handler(s) will be triggered concurrently to the processing of these event messages; this may be a problem for example if a controller listens to the `close_window` event emitted by a main frame in order to perform a proper termination: the basic, built-in event handlers will then by default be triggered whereas the controller teardown may be still in progress, and this may result in errors (ex: OpenGL ones, like `{{{badarg,"This"},{wxGLCanvas,swapBuffers,1}},...` because the built-in close handlers already deallocated the OpenGL context); a proper design is to ensure that `skip` remains set to false so that propagation of such events is disabled in these cases: then only the user code is in charge of closing the application, at its own pace[8]

- in terms of sizing, the dimensions of a parent widget prevail: its child widgets have to adapt (typically with sizers); if wanting instead that the size of a child dictates the one of its parent, the size of the client area of the parent should be set to the best size of its child, or its `fit/1` method shall be called

Extra information resources about `wx` (besides the documentation of its modules):

- wxErlang: Getting started and Speeding up, by Arif Ishaq
- Doug Edmunds' wxerlang workups
- wxWidgets itself

Micro-Cheat Sheet

To avoid having to perform a lookup in the documentation:

- Erlang indices start at `1`, except the ones of the `array` module that are zero-based
- for tuples of unknown number of elements:
  - **setting** an element is to be done with `setelement(positive_index(), tuple(), term()) -> tuple()`
  - **extracting** an element is to be done with `element(positive_index(), tuple()) -> term()`
  - **adding** an element at a given index (ex: to append a record tag) is to be done with `erlang:insert_element(positive_index(), tuple(), term()) -> tuple()`
  - **removing** an element at a given index (ex: to chop a record tag) is to be done with `erlang:delete_element(positive_index(), tuple()) -> tuple()`

(no need for the `erlang` module to be explicitly specified for the first two functions, as both are auto-imported)

- for `maps`: refer to map expressions, the maps module and the corresponding part of the efficiency guide; in short:
  - creating/updating maps is `EmptyMap = #{}`, `NewMFromScratch = #{K1 => V1, K2 => V2}` or `UpdatedM = M#{K1 => V1, K2 => V2}`

[7] MyriadGUI took a different convention: whether an event will propagate by default depends on its type, knowing that most of the types are to propagate. Yet the user can override these default behaviours, by specifying either the `trap_event` or the `propagate_event` subscription option, or by calling either the `trap_event/1` or the `propagate_event/1` function.

[8] This is why MyriadGUI applies per-event type defaults, thus possibly trapping events; in this case, if the built-in backend mechanisms would still be of use, they can be triggered by calling the `propagate_event/1` function from the user-defined handler, only once all its prerequisite operations have been performed (this is thus a way of restoring sequential operations).

  - updating the value associated to an already-existing key `K` is `ModM = M#{K := V}`
  - matching is `#{K1 := V1, K2 := V2} = M` where the `K*` are guard expressions and the `V*` can be any pattern

- when converting a float to a string, to set the precision (the number of digits after the decimal point) to `P`, use: `"~.Pf"`; for example, if `P=5`, we have: `"0.33333" = io_lib:format("~.5f", [1/3]).`
- the lesser-known "--" list subtraction operator returns a list that is a copy of the first argument where, for each element in the second argument, the first occurrence of this element (if any) is removed
- refer to the escape sequences, typically in order to specify characters like "space" (`$\s`); `$char` is the notation to designate the ASCII value or Unicode code-point of the character `char` (ex: `$A`)
- see the always handy operator precedence and the various ways to handle exceptions with try/catch
- in **guard expressions**:
  - to express an "and" operator, use a comma, i.e. `","`, like in `f(A, B, C, D) when A =:= B, C =:= A + D ->`
  - to express an "or" operator, use a semi-colon, i.e. `";"`
  - `andalso` and `orelse` behave there as tests, and their code is a little less effective than `","` and `";"`
  - refer to the two tables (test/others) of the BIFs supported in guards expressions
- in boolean expressions **outside of guards**:
  - do not use `and` and `or`, which are strict boolean operators, not intended for control; indeed their precedence is low (parentheses around the conditions being then necessary, like in `(A =:= B) and (C =:= D)`) and, more importantly, they do not short-circuit, i.e. in `E = A and B`, B will be evaluated even if A is already known as false (so is E), and in `E = A or B`, B will be evaluated even if A is already known as true (so is E)
  - prefer `andalso` and `orelse`, which are *control* operators; they short-circuit, and their precedence is low enough to spare the need of extra parentheses
  - so `case (A =:= B) and (C =:= D) of` shall become `case A =:= B andalso C =:= D of`
  - a common pattern is to use `andalso` in order *to evaluate a target expression iff a boolean expression is true*, to replace a longer expression like:

```
case BOOLEAN_EXPR of

    true ->
        DO_SOMETHING;

    false ->
        ok

end
```

with the equivalent (provided `BOOLEAN_EXPR` evaluates to either `true` or `false` - otherwise a `bad argument` exception will be thrown) yet shorter: `BOOLEAN_EXPR andalso DO_SOMETHING`; for example: `[...], OSName =:= linux andalso fix_for_linux(), [...]`

Similarly, `orelse` can be used *to evaluate a target expression iff a boolean expression is false*, to replace a longer expression like:

```
case BOOLEAN_EXPR of

    true ->
        ok;

    false ->
        DO_SOMETHING;

end
```

with: `BOOLEAN_EXPR orelse DO_SOMETHING`; for example: `[...], file_utils:exists("/etc/passwd") orelse throw(no_password_file), [...]`

In both `andalso / orelse` cases, the `DO_SOMETHING` branch may be a single expression, or a sequence thereof (i.e. a body), in which case a `begin/end` block may be of use, like in:

```
file_utils:exists("/etc/passwd") orelse
  begin
      trace_utils:notice("No /etc/password found."),
      throw(no_password_file)
  end
```

Similarly, taking into account the aforementioned precedences, `Count =:= ExpectedCount orelse throw({invalid_count, Count})` will perform the expected check.

Be wary of not having precedences wrong, lest bugs are introduced like the one in:

```
MaybeListenerPid =:= undefined orelse
   MaybeListenerPid ! {onDeserialisation, [self(), FinalUserData]}
```

(`orelse` having more priority than `!`, parentheses shall be added, otherwise, if having a PID, the message will actually be sent to any process that would be registered as `true`)

Some of these elements have been adapted from the Wings3D coding guidelines.

Erlang Resources

- the reference is the Erlang official website
- for teaching purpose, we would dearly recommend Learn You Some Erlang for Great Good!; many other high-quality Erlang books exist as well; one may also check the Erlang track on Exercism
- in addition to the module index mentioned in the Erlang Installation section, using the online search and/or Erldocs may also be convenient
- the Erlang community is known to be pleasant and welcoming to newcomers; one may visit the Erlang forums, which complement the erlang-questions mailing list (use this mirror in order to search through the past messages of this list)
- for those who are interested in parse transforms (the Erlang way of doing metaprogramming), the section about The Abstract Format is essential (despite not being well known)
- to better understand the inner workings of the VM: The Erlang Runtime System, a.k.a. "the BEAM book", by Erik Stenman
- BEAM Wisdoms, by Dmytro Lytovchenko

About 3D

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Monday, July 4, 2022

As usual, these information pertain to a GNU/Linux perspective.

Cross-Platform Game Engines

The big three are Godot, Unreal Engine and Unity3D.

Godot

Godot is our personal favorite engine, notably because it is free software (released under the very permissive MIT license).

See its official website and its asset library.

Godot (version 3.4.1) will not be able to load FBX files that reference formats like PSD or TIF and/or of older versions (ex: FBX 6.1). See for that our section regarding format conversions.

Installation On Arch Linux: `pacman -Sy godot`.

Use Godot logs are stored per-project; ex: `~/.local/share/godot/app_userdata/my-test-project/logs` past log files are kept once timestamped. They tend not to have interesting content.

A configuration tree lies in `.config/godot`, a cache tree in `~/.cache/godot`.

Unreal Engine

Another contender is the Unreal Engine, a C++ game engine developed by Epic Games; we have not used it yet.

The Unreal Engine 5 brings new features that may be of interest, including a fully dynamic global illumination and reflection system (Lumen, not requiring baked lightmaps anymore), a virtualized geometry system (Nanite, simplifying detailed geometries on the fly) and a quality 2D/3D asset library (the Quixel Megascans library, obtained from real-world scans).

Unreal does not offer a scripting language anymore, user developments have to be done in C++ (beyond *Blueprints Visual Scripting*).

Its licence is meant to induce costs only when making large-enough profits; more precisely, a *5% royalty is due only if you are distributing an off-the-shelf product that incorporates Unreal Engine code (such as a game) and the lifetime gross revenue from that product exceeds $1 million USD* (the first $1 million remaining royalty-exempt); in case of large success, it may be a costlier licence than Unity.

With an Unreal user account, the sources of the engine (in its latest stable version, 5) can be examined on Github (so it is open source, yet not free software).

See its official website and its marketplace.

Assets Purchased assets may be used in one's own shipped products (source) and apparently at least usually no restrictive terms apply.

Assets not created by Epic Games can be used in other engines unless otherwise specified (source).

Unity3D

Unity is most probably the cross-platform game engine that is the most popular.

Regarding the licensing of the engine, various plans apply, depending notably on whether one subscribes as an individual or a team, and on one's profile, revenue and funding; the general idea is not taking royalties, but **flat**, per seat yearly fees increasing with the organisation "size" (typically in the $400-$1800 range, per seat).

See its official website and its asset store.

Unity may be installed at least in order to access its asset store, knowing that apparently an asset purchased in this store may be used with any game engine of choice. Indeed, for the standard licence, it is stipulated in the EULA legal terms that:

Licensor grants to the END-USER a non-exclusive, worldwide, and perpetual license to the Asset to integrate Assets only as incorporated and embedded components of electronic games and interactive media and distribute such electronic game and interactive media.

So, in legal terms, an asset could be bought in the Unity Asset Store and used in Godot, for example - provided that its content can be used there technically without too much effort/constraints (this may happen with prefabs, specific animations, materials or shaders, conventions in use, etc.).

Installation Unity shall now be obtained thanks to the Unity Hub.

On Arch Linux it is available through the AUR, as an AppImage; one may thus use: `yay -Sy unityhub`.

Then, when running (as a non-priviledged user) `unityhub`, a Unity account will be needed, then a licence, then a Unity release will have to be added in order to have it downloaded and installed for good, covering the selected target platforms (ex: Linux and Windows "Build Supports").

We rely here on the Unity version 2021.2.7f1.

Additional information: Unity3D on Arch.

Configuration Configuring Unity so that its interface (mouse, keyboard bindings) behave like, for example, the one of Blender, is not natively supported.

Running Unity Just execute `unityhub`, which requires signing up and activating a licence.

Troubleshooting The log files are stored in `~/.config/unity3d`:

- Unity Editor: `Editor.log` (the most interesting one)
- Unity Package Manager: `upm.log`
- Unity Licensing client: `Unity.Licensing.Client.log`

If the editor is stuck (ex: when importing an asset), one may use as a last resort kill-unity3d.sh.

In term of persistent state, beyond the project trees themselves, there are:

- `~/.config/UnityHub/` and `~/.local/share/UnityHub/`
- `~/.config/unity3d/` and `~/.local/share/unity3d/`

(nothing in `~/.cache` apparently)

Unity Assets Once ordered through the Unity Asset Store, assets can be downloaded through the `Window -> Package Manager` menu, by replacing, in the top `Packages` drop-down, the `In Project` entry by the `My Assets` one. After having selected an asset, use the `Download` button at the bottom-right of the screen.

Then, to gain access to such downloaded assets, of course the simplest approach is to use the Unity editor; this is done by creating a project (ex: `MyProject`), selecting the aforementioned menu option (just above), then clicking on `Import` and selecting the relevant content that will end up in clear form in your project, i.e. in the UNIX filesystem with their actual name and content, for example in `MyProject/Assets/CorrespondingAssetProvider/AssetName`. We experienced reproducible freezes when importing resources.

Yet such Unity packages, once downloaded (whether or not they have been imported in projects afterwards) are files stored typically in the `~/.local/share/unity3d/Asset Store-5.x` directory and whose extension is `.unitypackage`.

Such files are actually `.tar.gz` archives, and thus their content can be listed thanks to:

```
$ tar tvzf Foobar.unitypackage
```

Inside such archives, each individual package resource is located in a directory whose name is probably akin to the checksum of this resource (ex: `167e85f3d750117459ff6199b79166fd`)[9]; such directory generally contains at least 3 files:

- `asset`: the resource itself, renamed to that unique checksum name, yet containing its exact original content (ex: the one of a Targa image)
- `asset.meta`: the metadata about that asset (file format, identifier, timestamp, type-specific settings, etc.), as an ASCII, YAML-like, text
- `pathname`: the path of that asset in the package "virtual" tree (ex: `Assets/Foo/Textures/baz.tga`)

When applicable, a `preview.png` file may also exist.

Some types of content are Unity-specific and thus may not transpose (at least directly) to another game engine. This is the case for example for materials or prefabs (whose file format is relatively simple, based on YAML 1.1).

Tools like AssetStudio (probably Windows-only) strive to automate most of the process of exploring, extracting and exporting Unity assets.

Meshes are typically in the FBX (proprietary) file format, that can nevertheless be imported in Blender and converted to other file formats (ex: gltF 2.0); see blender import and blender convert for that.

[9] Yet no checksum tool among `md5sum`, `sha1sum`, `sha256sum`, `sha512sum`, `shasum`, `sha224sum`, `sha384sum` seems to correspond; it must a be a different, possibly custom, checksum.

3D Data

File Formats

They are designed to store 3D content (scenes, nodes, vertices, normals, meshes, textures, materials, animations, skins, cameras, lights, etc.).

glTF We prefer to rely on the open, well-specified, modern glTF 2.0 format in order to perform import/export operations.

It comes in two forms:

- either as `*.gltf` when JSON-based, possibly embedding the actual data (vertices, normals, textures, etc.) as ASCII base64-encoded content, or referencing external files
- or as `*.glb` when binary; this is the most compact form, and the one that we recommend especially

See also the glTF 2.0 quick reference guide, the related section of Godot and this standard viewer of predefined glTF samples.

This (generic) online glTF viewer proved lightweight and convenient notably because it displays errors, warnings and information regarding the glTF data that it decodes.

Collada The second best choice that we see is Collada (`*.dae` files), an XML-based counterpart (open as well, yet older and with less validating facilities) to glTF.

FBX, OBJ, etc. Often, assets can be found as FBX of OBJ files and thus may have to be converted (typically to glTF), which is never a riskless task. FBX comes in two flavours: text-based (ASCII) or binary, see this retro-specification for more information.

In General Refer to blender import in order to handle the most common 3D file formats, and the next section about conversions.

The `file` command is able to report the version of at least some formats; for example:

```
# Means FBX 7.3:
$ file foobar.fbx
foobar.fbx: Kaydara FBX model, version 7300
```

Too often, some tool will not be able to load a file and will fail to properly report why. When suspecting that a binary file (ex: a FBX one) references external content either missing or in an unsupported format (ex: PSD or TIFF?), one may peek at their content without any dedicated tool, directly from a terminal, like in:

```
$ strings my_asset.fbx | sort | uniq | grep '\.'
```

This should list, among other elements, the paths that such a binary file is embedding.

Conversions

Due to the larger number of 3D file formats and the role of commercial software, interoperability regarding 3D content is poor and depends on many versions (of tools and formats).

Recommended Option: Relying on Blender Using blender import is the primary solution that we see: a content, once imported in Blender, can be saved in any of the supported formats.

Yet this operation may fail, for example on "older" FBX files, whose FBX version (ex: 6.1) is not supported by Blender ("*Version 6100 unsupported, must be 7100 or later*") or by other tools such as Godot. See also the Media Formats supported by Blender.

Workaround #1: Using Autodesk FBX Converter The simpler approach seems to download the (free) Autodesk FBX Converter and to use wine to run it on GNU/Linux. Just install then this converter with: `wine fbx20133_converter_win_x64.exe`.

A convenient alias (based on default settings, typically to be put in one's `~/.bashrc`) can then be defined to run it:

```
$ alias fbx-converter-ui="$HOME/.wine/drive_c/Program\ Files/Autodesk/FBX/FBX\ Convert
```

Conversions may take place from, for example, FBX 6.1 (also: 3DS, DAE, DXF, OBJ) to a FBX version in: 2006, 2009, 2010, 2011, 2013 (i.e. 7.3 - of course the most interesting one here), but also DXF, OBJ and Collada, with various settings (embedded media, binary/ASCII mode, etc.).

An even better option is to use directly the command-line tool `bin/FbxConverter.exe`, which the previous user interface actually executes. Use its `/?` option to get help, with interesting information.

For example, to update a file in a presumably older FBX into a 7.3 one (that Blender can import):

```
$ cd ~/.wine/drive_c/Program\ Files/Autodesk/FBX/FBX\ Converter/2013.3/bin
$ FbxConverter.exe My-legacy.FBX newer.fbx /v /sffFBX /dffFBX /e /f201300
```

We devised the update-fbx.sh script to automate such an in-place FBX update.

Unfortunately, at least on one FBX sample taken from a Unity package, if the mesh could be imported in Blender, textures and materials were not (having checked `Embed media` in the converter or not).

Workaround #2: Relying on Unity Here the principle is to import a content in Unity (the same could probably be done with Godot), and to export it from there.

Unity does not allow to export for example FBX natively, however a package for that is provided. It shall be installed first, once per project.

One shall select in the menu `Window -> Package Manager`, ensure that the entry `Packages:` points to `Unity Registry`, and search for `FBX Exporter`, then install it (bottom right button).

Afterwards, in the `GameObject` menu, an `Export to FBX` option will be available. Select the `Binary` export format (not `ASCII`) if wanting to be compliant with Blender.

Samples

Here are a few samples of 3D content (useful for testing):

- glTF, notably glTF 2.0; direct: .gltf Buggy example, .glb Fish example (also: a simple cube)
- DAE; direct: Duck example (also: a simple cube)
- FBX; direct: Stylized character
- OBJ
- IFC; direct: Basic house (requires the BlenderBIM add-on for BIM support in Blender)

Asset Providers

Usually, for one's creation, much multimedia artwork has to be secured: typically graphical assets (ex: 2D/3D geometries, animations, textures) and/or audio ones (ex: music, sounds, speech syntheses, special effects).

Instead of creating such content by oneself (not enough time/interest/skill?), it may be more relevant to rely on specialised third-parties.

Hiring a professional or a freelance is then an option. This is of course relatively expensive, involves more efforts (to define requirements and review the results), longer, but it is to provide exactly the artwork that one would like.

Another option is to rely on specialised third-party providers that **sell non-exclusive licences for the content they offer**.

These providers can be either direct **content producers** (companies with staffs of modellers), or **asset aggregators** (marketplaces which federate the offers of many producers of any size) that are often created in link to a given multimedia engine. An interesting point is that assets purchased in these stores generally can be used in any technical context, hence are not meant to be bound to the corresponding engine.

Nowadays, much content is available, in terms of theme/setting (ex: Medieval, Science-Fiction, etc.), of nature (ex: characters, environments, vehicles, etc.), etc. and the overall quality/price ratio is rather good.

The main advantages of these marketplaces is that:

- they favor the competition between content providers: the clients can easily compare assets and share their opinion about them
- they generalised simple, standard, unobtrusive licensing terms; ex: royalty free, allowing content to be used as they are or in a modified form, not limited by types of usage, number of distributed copies, duration of use,

number of countries addressed, etc.; the general rule is that much freedom is left to the asset purchasers provided that they use for their own projects (rather than for example selling the artwork as they are)

The main content aggregators that we spotted are (roughly by decreasing order of interest, based on our limited experience):

- the Unity Asset Store, already discussed in the Unity Assets section; websites like this one allow to track the significant discounts that are regularly made on assets
- the UE Marketplace, i.e. the store associated to the Unreal Engine; in terms of licensing and uses:
  - this article states that *When customers purchase Marketplace products, they get a non-exclusive, worldwide, perpetual license to download, use, copy, post, modify, promote, license, sell, publicly perform, publicly display, digitally perform, distribute, or transmit your product's content for personal, promotional, and/or commercial purposes. Distribution of products via the Marketplace is not a sale of the content but the granting of digital rights to the customer.*
  - this one states that *Any Marketplace products that have not been created by Epic Games can be used in other engines unless otherwise specified.*
  - this one states that *All products sold on the Marketplace are licensed to the customer (who may be either an individual or company) for the lifetime right to use the content in developing an unlimited number of products and in shipping those products. The customer is also licensed to make the content available to employees and contractors for the sole purpose of contributing to products controlled by the customer.*
- itch.io
- Turbosquid
- Free3D
- CGtrader
- ArtStation
- Sketchfab
- 3DRT
- Reallusion
- Arteria3D
- the GameDev Market (GDM)
- the Game Creator Store

Many asset providers organise interesting discount offers (at least -50% on a selection of assets, sometimes even more for limited quantities) for the Black Friday (hence end of November) or for Christmas (hence mid-December till the first days of January).

Modelling Software

Blender

Blender is a very powerful open-source 3D toolset.

Blender (version 3.0.0) can import FBX files of version at least 7.1 ("7100"). See for that our section regarding format conversions.

We recommend the use our Blender scripts in order to:

- import conveniently various file formats in Blender, with `blender-import.sh`
- convert directly on the command-line various file formats (still thanks to a non-interactive Blender), with `blender-convert.sh`

Wings3D

Wings3D is a nice, Erlang-based, free software, advanced subdivision *modeler*[10], available for Windows, Linux, and Mac OS X. Wings3D relies on OpenGL.

It can be installed on Arch Linux, from the AUR, as `wings3d`; one can also rely on our Wings3D scripts in order to install and/or execute it.

We prefer using the Blender-like camera navigation conventions, which can be set in Wings3D by selecting `Edit -> Preferences -> Camera -> Camera Mode` to `Blender`.

See also:

- its official website
- its development project
- its build instructions for UNIX-like systems

Other Tools

Draco

Draco is an open-source library for compressing and decompressing 3D geometric meshes and point clouds.

It is intended to improve the storage and transmission of 3D graphics; it can be used with glTF, with Blender, with Compressonator, or separately.

A `draco` AUR package exists, and results notably in creating the `/usr/lib/libdraco.so` shared library file.

Even once this package is installed, when Blender exports a mesh, a message like the following is displayed:

[10] As opposed to *renderer*; yet Wings3D integrates an OpenCL renderer as well, deriving from LuxCoreRender, an open-source Physically Based Renderer (it simulates the flow of light according to physical equations, thus producing realistic images of photographic quality).

```
'/usr/bin/3.0/python/lib/python3.10/site-packages/libextern_draco.so' does
not exist, draco mesh compression not available, please add it or create
environment variable BLENDER_EXTERN_DRACO_LIBRARY_PATH pointing to the folder
```

Setting the environment prior to running Blender is necessary (and done by our `blender-*.sh` scripts:

```
$ export BLENDER_EXTERN_DRACO_LIBRARY_PATH=/usr/lib
```

but not sufficient, as the built library does not bear the expected name.
So, as root, one shall fix that once for all:

```
$ cd /usr/lib
$ ln -s libdraco.so libextern_draco.so
```

Then the log message will become:

```
'/usr/lib/libextern_draco.so' exists, draco mesh compression is available
```

The Compressonator

The Compressonator is an AMD tool (as a GUI, a command-line executable and a SDK) designed to compress textures (ex: in DXT1, DXT3 or DXT5 formats; typically resulting in a `.dds` extension) and generate mipmaps ahead of time, so that it does not have to be done at runtime.

F3D

f3d (installable from the AUR) is a fast and minimalist VTK-based 3D viewer.

Such a viewer is especially interesting to investigate whether a tool failed to properly export a content or whether it is the next tool that actually failed to properly import, and to gain another chance to have relevant error messages.

OpenGL Corner

Conventions

Refer to our Mini OpenGL Glossary for most of the terms used in these sections.

Code snippets will corresponds to the OpenGL/GLU APIs as they are exposed in Erlang, in the gl and glu modules respectively.

These translate easily for instance in the vanilla C GL/GLU implementations. As an example, gl:ortho/6 (`6` designating here the arity of that function, i.e. the number of the arguments that it takes) corresponds to its C counterpart, glOrtho.

The reference pages for OpenGL (in version 4.x) can be browsed here.

Note that initially the information here related to older versions of OpenGL (1.1, 2.1, etc.; see history) that relied on a fixed pipeline (no shader support) - whereas, starting from OpenGL 3.0, many of the corresponding features were marked as deprecated, and actually removed in 3.1. However, thanks to the

compatibility context (whose support is not mandatory - but that all major implementations of OpenGL provide), these features can still be used.

Yet nowadays relying on at least OpenGL 3 *core* context (not using the compatibility context) would be preferable (source: this thread). Still better options would be to rely on OpenGL 4 Core or OpenGL ES 2+, or libraries on top of Vulkan, like wgpu. Specific libraries also exist for rendering for the web and for mobile, like WebGPU.

As of 2022, the current OpenGL version is 4.6; we will try to stick to the latest ones (4.x) only (ex: skipping intermediate changes in 3.2); even though in this document reminiscences of older OpenGL versions remain, the current minimum that we target is the **Core Profile of OpenGL 3.3**, which is "modern OpenGL" and introduced most features that still apply (and will halt on error if any deprecated functionality is used).

For more general-purpose computations (as opposed to rendering operations) to be offset to a GPU/GPGU, one may rely on OpenCL instead.

The mentioned tests will be Ceylan-Myriad ones, typically located here.

Basics

- OpenGL is a **software interface to graphics hardware**, i.e. the *specification* of an API (of around 150 functions in its older version 1.1), developed and maintained by the Khronos Group

- a video card will run an *implementation* of that specification, generally developed by the manufacturer of that card; a good rule of thumb is to always update one's video card drivers to its **latest stable version**, as OpenGL implementations are constantly improved (bug-fixing) and updated (with regard to newer OpenGL versions)

- OpenGL concentrates on **hardware-independent 2D/3D rendering**; no commands for performing window-related tasks or obtaining user input are included; for example frame buffer configuration is done outside of OpenGL, in conjunction with the windowing system

- OpenGL offers only **low-level primitives** organised through a pipeline in which vertices are assembled into primitives, then to fragments, and finally to pixels in the frame buffer; as such, OpenGL is a building-block for higher-level engines (ex: like Godot)

- OpenGL is a **procedural** (function-based, not object-oriented) **state machine** comprising a larger number of variables defined within a given OpenGL state (named *OpenGL context*; comprising vertex coordinates, textures, frame buffer, etc.); said otherwise, relatively to an OpenGL context (which is often implicit), all OpenGL state variables behave like global variables; when a parameter is set, it applies and lasts as long as it is not modified; the effect of an OpenGL command may vary depending on whether certain modes are enabled (i.e. whether some state variables are set)

- so the **currently processed element** (ex: a vertex) **inherits (implicitly) the current settings of the context** (ex: color, normal, texture coordinate, etc.); this is the only reasonable mode of operation, knowing

that a host of parameters apply when performing a rendering operation (specifying all these parameters would not be a realistic option); as a result, any specific parameter shall be set first (prior to triggering such an operation), and is to last afterwards (being "implicitly inherited"), until possibly being reassigned in some later point in time

- OpenGL respects a **client/server execution model**: an application (a specific client, running on a CPU) issues commands to a rendering server (on the same host or not - see GLX; generally the server can be seen as running on a local graphic card), that executes them **sequentially and in-order**; as such, most of the calls performed by user programs are **asynchronous**: they are triggered by the program through OpenGL, and return almost immediately, whereas they have not been executed yet; they have just be queued; indeed OpenGL implementations are almost always pipelined, so the rendering must be thought as primarily taking place in a background process; additional facilities like *Display Lists* allow to pipeline operations (as opposed to the default *immediate mode*), which are accumulated for processing at a later time, as fast as the graphic card can process them

- state variables are mostly server-side, yet some of them are client-side; in both cases, they can be gathered in *attribute groups*, which can be pushed on, and popped from, their respective server or client attribute stacks

- OpenGL manages two types of data, handled by mostly different paths of its rendering pipeline, yet that are ultimately integrated in the framebuffer through fragment-yielding rasterization:

  - geometric data (vertices, lines, and polygons)
  - pixel data (pixels, images, and bitmaps)

- vertices and normals are transformed by the **model-view** and **projection** matrices (that can be each set and transformed on a stack of their own), before being used to produce an image in the frame buffer; texture coordinates are transformed by the **texture** matrix

- textures may reside in the main, general-purpose, client, **CPU-side memory** (large and slow to access for the rendering) and/or in any auxiliary, dedicated, server-side **GPU memory** (more constrained, hence prioritized thanks to *texture objects*; and, rendering-wise, of significantly higher-performance)

- OpenGL has to apply any kind of transformation, linear (ex: rotation, scaling) or not (ex: translation, perspective) to geometries, for example in order to perform referential changes or rendering; each of these transformations can be represented as a 4x4 homogeneous matrix, with floating-point (homogeneous) coordinates[11]; a series of transformations can then simply be represented as a single of such matrices, corresponding to the product of the involved transformation matrices

- while this will not change anything regarding the actual OpenGL library and the computations that it performs, the conventions adopted by the OpenGL *documentation* regarding matrices are the following ones:
  - their in-memory representation is column-major order (even if it is unusual, at least in C; this corresponds to Fortran-like conventions), meaning that it enumerates their coordinates first per column rather than per row (and for them a vector is a *row* of coordinates), whereas tools following the row-major counterpart order, including Myriad (precisely: MyriadGUI), do the opposite (and vectors are *columns* of coordinates); more clearly, a matrix like $M = \begin{bmatrix} a11 & a12 & ... & a1n \\ a21 & a22 & ... & a2n \\ ... & ... & ... & ... \\ am1 & am2 & ... & amn \end{bmatrix}$
    - will be stored with row-major conventions (ex: Myriad) as: `[ [a11, a12, ... a1n], [a21, a22, ... a2n], ..., [am1, am2, ... amn] ]`
    - whereas, with the conventions discussed, OpenGL will expect it to be stored in-memory in this order: `a11, a21, ..., am1, a21, a22, ..., am2, ..., a1n, a2n, ..., amn`, i.e. as the transpose of the previous matrix
  - these OpenGL storage conventions do not tell how matrices are to be multiplied (knowing of course that the matrix product is not commutative); if following the aforementioned OpenGL *documentation* conventions, one should consider that OpenGL relies on the usual multiplication order, that is **post-multiplication**, i.e. *multiplication on the right*; this means that, if applying on a given matrix M a transformation O (ex: rotation, translation, scaling, etc.) represented by a matrix M_O, the resulting matrix will be $M' = M.M_O$ (and not $M' = M_O.M$); a series of operations O_1, then O_2, ..., then O_n will therefore correspond to a matrix $M' = M_{O1}.M_{O2}.[...].M_{On}$; applying a vector $\vec{V}$ to a matrix M will result in $\vec{V'} = M.\vec{V}$
  - so when an OpenGL program performs calls like first for a rotation (r), then for a scaling (s) and finally for a translation (t):

```
glRotatef(90, 0, 1, 0);
glScalef(1, 10, 1);
glTranslatef(5,10,5);
```

[11] So a 3D point is specified based on 4 coordinates: $P = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$, with `w` being usually equal to `1.0` (otherwise the point can be normalised by dividing each of its coordinates by `w`, provided of course `w` is not null - otherwise these coordinates do not specify a point but a direction).

Thus summing (like vectors) two 4D points actually returns their mid-point (center of segment), as `w` will be normalised: $P1+P2 = \begin{pmatrix} x1 \\ y1 \\ z1 \\ 1.0 \end{pmatrix} + \begin{pmatrix} x2 \\ y2 \\ z2 \\ 1.0 \end{pmatrix} = \begin{pmatrix} x1+x2 \\ y1+y1 \\ z1+z2 \\ 2.0 \end{pmatrix} = \begin{pmatrix} (x1+x2)/2.0 \\ (y1+y2)/2.0 \\ (z1+z2)/2.0 \\ 1.0 \end{pmatrix}$

the current matrix M ends up being multiplied (on the right) by $M' = M_r.M_s.M_t$; when applied to a vector $\vec{V}$, still multiplying on the right results in $\vec{V'} = M.\vec{V} = M.M_r.M_s.M_t.\vec{V'}$; so the input vector $\vec{V}$ is first translated, then the result is scaled, then rotated, then transformed by the previous matrix M; as a result: **operations happen in the opposite order of their specification as calls**; said differently: one shall specify the calls corresponding to one's target series of transformations *backwards*

  - considering that the OpenGL storage is done in a surprising column-major order was actually a trick so that OpenGL could rely on the (modern, math-originating) vector-as-column convention while being still compliant with its GL ancestor - which relied on the (now unusual) vector-as-row convention and on *pre-multiplication* (where we would have $M' = M_O.M$); indeed, knowing that, when transposing matrices, $(A.B)^\top = B^\top.A^\top$, one may consider that OpenGL actually always operates on transpose elements, and thus that: (1) matrices are actually specified in row-order and (2) they are multiplied on the left (ex: $M' = M_t.M_s.M_r.M$); note that switching convention does not affect at all the computations, and that the same operations are always performed in reverse call order

- OpenGL can operate on three mutually exclusive **modes**:
  - *rendering*: the default, most common one, discussed here
  - *feedback*: allows to capture the primitives generated by the vertex processing, i.e. to establish the primitives that would be displayed after the transformation and clipping steps; often used in order to resubmit this data multiple times
  - *selection*: determines which primitives would be drawn into some region of a window (like in *feedback* mode), yet based on stacks of only user-specified "names" (so that the actual data of the corresponding primitives is not returned, just their name identifier); a special case of selection is *picking*, allowing to determine what are the primitives rendered at a given point of the viewport (typically the onscreen position of the mouse cursor, to enable corresponding interactions)

Steps for OpenGL Rendering The usual analogy to describe them is the process of **producing a photography**:

1. a set of elements (3D objects) can be placed (in terms of position and orientation) as wanted in order to compose one's scene of interest (*modelling transformations*, with world coordinates)
2. the photographer may similarly place as wanted his camera (*viewing transformations*, with camera coordinates)
3. the settings of the camera can be adjusted, for example regarding its lens / zoom factor (*projection transformations*, with window coordinates)
4. the snapshots that it takes can be further adapted before being printed, for example in terms of scaling (*viewport transformations*, with screen coordinates)

One can see that the first two steps are reciprocal; for example, moving all objects in a direction or moving the camera in the opposite direction is basically the same operation. These two operations, being the two sides of the same coin, can thus be managed by a single matrix, the *model-view* one.

Finally, as mentioned in the section about storage conventions, in OpenGL, operations are to be defined in reverse order. If naming M_s the matrix implementing a given step S, the previous process would be implemented by an overall matrix, based on the previous bullet numbers: $M = M_4.M_3.M_2.M_1$, so that applying a vector $\vec{V}$ to M results in $\vec{V'} = M.\vec{V} = M_4.M_3.M_2.M_1.\vec{V} = M_4.(M_3.(M_2.(M_1.\vec{V})))$.

Transformations In this context, except notably the projections, most are invertible, and a composition of invertible transformations, in any combination and sequence, is itself invertible.

As mentioned, they can all be expressed as 4x4 homogeneous matrices, and their composition translates into the (orderly) product of their matrices.

Referential transitions are discussed further in this document, in the 3D referentials section.

Translations / Rotations / Scalings / Shearings

- the inverse of a **translation** of a vector $\vec{T}$ is a translation of vector $-\vec{T}$, thus: $(Mt_{\vec{T})^{-1}=Mt_{-\vec{T}}}$
- the inverse of a **rotation** of an angle θ along a vector $\vec{U}$ is a rotation of an angle $-\theta$ along the same vector, thus: $(Mr_(\vec{u},\theta))^{-1} = Mr_(\vec{u},-\theta)$
- the inverse of a **scaling** of a (non-null) factor f is a scaling of factor $1/f$, thus: $(Ms_f)^{-1} = Ms_{1/f}$; the same applies for each factor when performing a shear mapping

Reflections Symmetries with respect to an axis correspond to a scaling factor of -1 along this axis, and 1 along the other axes.

Affine Transformations An affine transformation designates all geometric transformations that preserve lines and parallelism (but not necessarily distances and angles).

They are compositions of a linear transformation and a translation of their argument.

For them $f(\lambda.x + y) = \lambda.f(x) + f(y)$.

Projections A projection defines 6 clipping planes (and at least 6 additional ones can be defined).

A 3D plane is defined by including a given (3D) point and comprising all vectors orthogonal to a given vector; it can be defined thanks to 4 coordinates (ex: `(a, b, c, d)`); and a point $P = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ will belong to such a plane iff $a.x + b.y + c.z + d = 0$.

Two kinds of projections are considered: orthographic and perspective.

Orthographic Projections

Their viewing volume is a parallelepiped, precisely a rectangular cuboid.

With such projections, parallel lines remain parallel; see `gl:ortho/6` and `glu:ortho2D/4`.

Perspective Projections

Their viewing volume is a truncated pyramid.

They are defined based on a field of view and an aspect ratio; see `gl:frustum/6` and `glu:perspective/4`.

Viewport Transformations

As for the viewport, it is generally defined with `gl:viewport/4` so that its size corresponds to the widget in which the rendering is to take place.

To avoid distortion, its aspect ratio must be the same as the one of the projection transformation.

Camera The default model-view matrix is an identity; the camera (or eye) is located at the origin, points down the negative Z-axis, and has an up-vector of `(0, 1, 0)`.

With Z-up conventions (like in Myriad ones), this corresponds to a camera pointing downward.

Calling `glu:lookAt/9` allows to set arbitrarily one's camera position and orientation.

In order to switch from (OpenGL) Y-up conventions to Z-up ones, another option is to rotate the initial (identity) model-view matrix along the X axis of an angle of $-\pi/2$, or to (post-)multiply the model-view matrix with:

$$M_{camera} = P_{zup \to yup} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, if we want that this camera sees, in (Z-up) MyriadGUI referential, a point `P` at coordinates $P_{zup} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ (thus a point in its Y axis), its coordinates in the base OpenGL (Y-up) referential must be $P_{yup} = P_{zup \to yup}.P_{zup} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$; refer to the Computing Transition Matrices section for more information.

Hints

- a frequent pattern is, for some type of OpenGL element (let's name it `Foo`; it could designate for example `Texture`, `Buffer` or `VertexArray`) is: first `glGenObjects(1, &fooId);` (note the plural) is used, then `glBindObject(GL_SOME_TARGET, fooId);`
  - it must be understood that `glGenObjects` is the actual creator of (at least) one new (blank) instance of `Foo`, whose address is kept

by OpenGL behind the scenes; the user program will be able to access this instance only once bound thanks to an additional level of indirection, its (GL) identifier (`fooId` here)

  - as for `glBindObject`, its role is to register the `Foo` pointer corresponding to the specified identifier `fooId` in the C-like struct that corresponds to the current context (i.e. the current state of OpenGL), in the field designated here by `GL_SOME_TARGET`, like in: `current_gl_context->gl_some_target = foo_pointer_for(fooId)`; this operation is thus mostly an assignment
  - once bound, this `Foo` instance can be accessed implicitly (through the current context) by calls such as `glSetFooOption(GL_SOME_TARGET, GL_OPTION_FOO_WIDTH, 800);` (where neither its identifier nor any pointer for it is specified); once done, this instance can be unbound with `glBindObject(GL_SOME_TARGET, 0);`; rebinding that identifier later will restore the corresponding options; as a result, several instances can be created, corresponding to as many sets of predefined options, and when a given one shall apply, it just has to be bound

- in OpenGL, the **alpha** coordinate encodes **opacity**; thus 1.0 means fully opaque, whereas 0.0 means fully transparent

Mini OpenGL Glossary Terms that are more or less specific to OpenGL:

- `Accumulation buffer`: a buffer that may be used for scene antialiasing; the scene is rendered several times, each time jittered less than one pixel, and the images are accumulated and then averaged
- `Alpha Test`: to reject fragments based on their alpha coordinate; useful to reduce the number of fragments rendered through transparent surfaces
- `Context`: a rendering context corresponds to the OpenGL state and the connection between OpenGL and the system; in order to perform rendering, a suitable context must be current (i.e. bound, active for the OpenGL commands); it is possible to have multiple rendering contexts share buffer data and textures, which is specially useful when the application uses multiple threads for updating data into the memory of the graphics card
- `DDS`: a file format suitable for texture compression that can be directly read by the GPU
- `Display list`: a series of OpenGL commands, identified by an integer, to be stored, server-side, for subsequent execution; it is defined so that it can be sent and processed more efficiently, and probably multiple times, by the graphic card (compared to doing the same in immediate mode)
- (pixel) `fragment`: two-dimensional description of elements (point, line segment, or polygon) produced by the rasterization step, before being stored as pixels in the frame buffer; also defined as: "a point and its associated information"; a fragment translates to a pixel after a process involving in turn: texture mapping, fog effect, antialiasing, tests (scissor, alpha, stencil, depth), blending, dithering, and logical operations on fragments (and, or, xor, not, etc.)

- `Evaluator`: the part of the pipeline to perform polynomial mapping (basis functions) and transform higher-level primitives (such as NURBS) into actual ones (vertices, normals, texture coordinates and colors)
- `Frame buffer`: the "server-side" pixel buffer, filled, after rasterization took place, by combinations (notably blending) of the selected fragments; it is actually made of a set of logical buffers of bitplanes: the color (itself comprising multiple buffers), depth (for hidden-surface removal), accumulation, and stencil buffers
- `GL`: *Graphics Library* (also a shorthand for *OpenGL*, which is an open implementation thereof)
- `GLU`: *OpenGL Utility Library*, a standard part of every OpenGL implementation, providing auxiliary features (ex: image scaling, automatic mipmapping, setting up matrices for specific viewing orientations and projections, performing polygon tessellation, rendering surfaces, supporting quadrics routines that create spheres, cylinders, cones, etc.); see this page for more information
- `GLUT`, *OpenGL Utility Toolkit*, a system-independent window toolkit hiding the complexities of differing window system APIs and more complicated three-dimensional objects such as a sphere, a torus, and a teapot; its main interest was when learning OpenGL, it is less used nowadays
- `GLX`: the X extension of the OpenGL interface, i.e. a solution to integrate OpenGL to X servers; see this page for more information
- `GLSL`: OpenGL Shading Language, a C-like language with which the transformation and fragment shading stages of the pipeline can be programmed; introduced in OpenGL 2.0; see our GLSL section
- `OpenCL`: Open Computing Language, a framework for writing programs that execute across heterogeneous platforms: central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators; in practice OpenCL defines programming languages, deriving from C and C++, for these devices, and APIs to control the platform and execute programs on the compute devices; OpenGL defines a standard interface for parallel computing using task- and data-based parallelism; see also our Erlang-related section
- `OpenGL ES`: OpenGL for Embedded Systems is a subset of the OpenGL API, designed for embedded systems (like smartphones, tablet computers, video game consoles and PDAs)
- `Pixel`: *Picture Element*
- `Primitive`: points, lines, polygons, images, and bitmaps
- (geometric) `Primitives`: they are (exactly) points, lines, and polygons
- `Rasterization`: the process by which a primitive is converted to a two-dimensional image

- `Scissor Test`: an arbitrary screen-aligned rectangle outside of which fragments will be discarded; useful to clear or update only a part of the viewport
- `Shader`: a user-defined program providing the code for some programmable stages of the rendering pipeline; they can also be used in a slightly more limited form for general, on-GPU computation (source)
- `Stencil Test`: conditionally discards a fragment based on the outcome of a selected comparison between the value in the stencil buffer and a reference value; useful to perform non-rectangular clipping
- `Texel`: *Texture Element* ; it corresponds to a `(s,t)` pair of coordinates in `[0,1]` designating a point in a texture
- `Vertex Array`: these in-memory client-side arrays may aggregate 6 types of data (vertex coordinates, RGBA colors, color indices, surface normals, texture coordinates, polygon edge flags), possibly interleaved; such arrays allow to reduce the number of calls to OpenGL functions, and also to share elements (ex: vertices pertaining to multiple faces should preferably be defined only once); in a non-networked setting, the GPU just dereferences the corresponding pointers
- `Viewport`: the (rectangular) part (defined based on its lower left corner and its width and height, in pixels) within the current window in which OpenGL is to perform its rendering; so multiple viewports may be used in turn in order to offer multiple, composite views of the scene of interest in a given window; the ultimately processed 2D coordinates in OpenGL are both in `[-1.0, 1.0]` before they are finally mapped to the current viewport dimensions (ex: abscissa in `[0,800]`, ordinate in `[0,600]`, in pixels)
- `Vulkan`: a low-overhead, cross-platform API, open standard for 3D graphics and computing; it is intended to offer higher performance and more balanced CPU and GPU usage than the OpenGL or Direct3D 11 APIs; it is lower-level than OpenGL, and not backwards compatible with it (source)
- `VAO`: a (GLSL) *Vertex Array Object* (OpenGL 4.x), able to store multiple VBOs (up to one for vertices, the others for per-vertex attributes); a VAO corresponds to an homogeneous chunk of data, sent from the CPU-space in order to be stored in the GPU-space; more information
- `VBO`: a (GLSL) *Vertex Buffer Object*, a buffer storing a piece of information (vertex coordinates, or normal, or colors, or texture coordinates, etc.) for each element of a series of vertices; more information

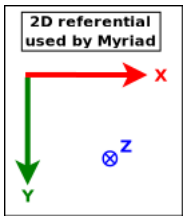
Refer to the description of the pipeline for further details.

Referentials

Referentials In 2D A popular convention, for example detailed in this section of the Red book, is to consider that the ordinates increase when going from the *bottom of the viewport to its top*; then for example the on-screen

lower-left corner of the OpenGL canvas is `(0,0)`, and its upper-right corner is `(Width,Height)`.

As for us, we prefer the MyriadGUI 2D conventions, in which ordinates increase when going from the *top of the viewport to its bottom*, as depicted in the following figure:



Such a setting can be obtained thanks to (with Erlang conventions):

```
gl:matrixMode(?GL_PROJECTION),
gl:loadIdentity(),

% Like glu:ortho2D/4:
gl:ortho(_Left=0.0, _Right=float(CanvasWidth),
  _Bottom=float(CanvasHeight), _Top=0.0, _Near=-1.0, _Far=1.0)
```

In this case, the viewport can be addressed like a **usual (2D) framebuffer** (like provided by any classical 2D backend such as SDL) obeying the coordinate system just described: if the width of the OpenGL canvas is 800 pixels and its height is 600 pixels, then its top-left on-screen corner is `(0,0)` and its bottom-right one is `(799,599)`, and any pixel-level operation can be directly performed there "as usual". One may refer, in Myriad, to `gui_opengl_2D_test.erl` for a full example thereof, in which line-based letters are drawn to demonstrate these conventions.

Each time the OpenGL canvas is resized, this projection matrix will have to be updated, with the same procedure, yet based on the new dimensions.

Another option - still with axes respecting the Myriad 2D conventions - is to operate this time based on **normalised, definition-independent coordinates**, ranging in `[0.0, 1.0]`, like in:

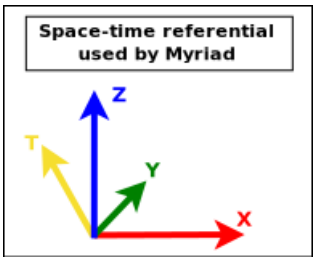
```
gl:matrixMode(?GL_PROJECTION),
gl:loadIdentity(),

gl:ortho(_Left=0.0, _Right=1.0, _Bottom=1.0, _Top=0.0, _Near=-1.0, _Far=1.0)
```

Using "stable", device-independent floats instead of integers directly accounting for pixels may be more convenient. For example a resizing of the viewport will then not require an update of the projection matrix. One may refer to `gui_opengl_minimal_test.erl` for a full example thereof.

Referentials In 3D We will rely here as well on the Myriad conventions, this time for 3D (not taking specifically time into account here):

These are thus Z-up conventions (the Z axis being vertical and designating altitudes), like modelling software such as Blender.



A Tree of Referentials In the general case, either in 2D or (more often of interest here) in 3D, a given scene (a model) is made of a set of elements (ex: the model of a street may comprise a car, two bikes, a few people) that will have to be rendered from a given viewpoint (ex: a window on the second floor of a given building) onto the (flat) user screen (with suitable clipping, perspective division and projection on the viewport). Let's start from the intended result and unwind the process.

The rendering objective requires to have ultimately one's scene transformed as a whole in eyes coordinates (to obtain coordinates along the aforementioned 2D screen referential, along the X and Y axes - the Z one serving to sort out depth, as per our conventions).

For that, a prerequisite is to have the target scene correctly composed, with all its elements defined in the same (scene-global) space, in their respective position and orientation (then only the viewpoint, i.e. the virtual camera, can take into account the scene as a whole, to transform it to eye coordinates).

As each individual type of model (ex: a bike model) is natively defined in an abstract, local referential (an orthonormal basis) of its own, each actual model instance (ex: the first bike, the second bike) has to be specifically placed in the referential of the overall scene. This placement is either directly defined in that target space (ex: bike A is at this absolute position and orientation in the scene global referential) or relatively to a *series* of parent referentials (ex: this character rides bike B - and thus is defined relatively to it, knowing that the bike is placed relatively to the car, and that the car itself is placed relatively to the scene).

So in the general case, referentials are nested (recursively defined relatively to their parent) and form a tree[12] whose root corresponds to the (possibly absolute) referential of the overall scene, like in:

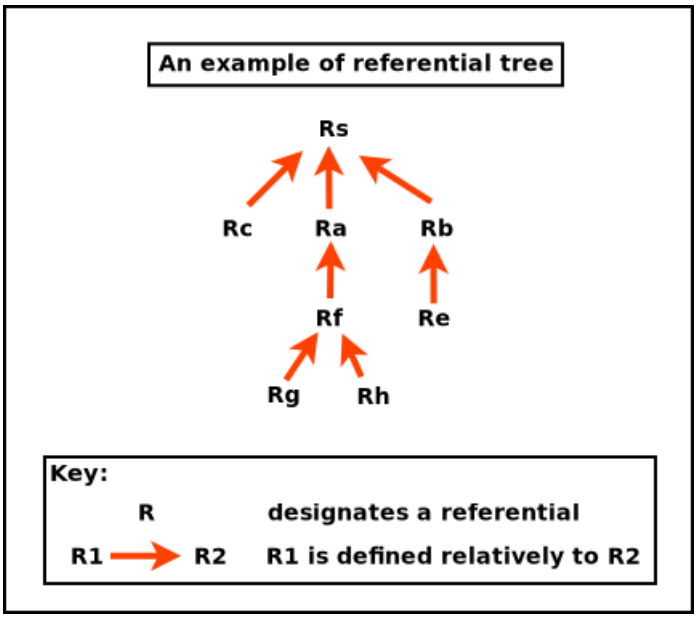
A series of model transformations has thus to be operated in order to express all models in the scene referential:

```
(local referential of model Rh) -> (parent referential Rf) -> (parent referential Ra)
```

For example the hand of a character may be defined in R_h, itself defined relatively to its associated forearm in R_f up to the overall referential R_a of that character, defined relatively to the referential of the whole scene, R_s. This

[12]This is actually named a *scene graph* rather than a *scene tree*, as if we consider the leaves of that "tree" to contain actual geometries (ex: of an abstract bike), as soon as a given geometry is instantiated more than once (ex: if having 2 of such bikes in the scene), this geometry will have multiple parents and thus the corresponding scene will be a graph.

As for us, we consider *referential trees* (no geometry involved) - a given 3D object being possibly associated to (1) a referential and (2) a geometry (independently).



referential may have no explicit parent defined, meaning implicitly that it is defined in the canonical, global referential.

Once the **model** is expressed as a whole in the scene-global referential, the next transformations have to be conducted : **view** and projection. The view transformation involves at least an extra referential, the one of the camera in charge of the rendering, which is R_c, possibly defined relatively to R_s.

So a geometry (ex: a part of the hand, defined in R_f) has been transformed upward in the referential tree in order to be expressed in the common, "global" scene referential R_s, before being transformed last in the camera one, R_c.

In practice, all these operations can be done thanks to the multiplication of homogeneous 4x4 matrices, each able to express any combination of rotations, scalings/reflections/shearings, translations, which thus include the transformation of one referential into another. Their product can be computed once, and then applying a vector (ex: corresponding to a vertex) to the resulting matrix allows to perform in one go the full composition thereof, encoding all model-view transformations and even the projection as well.

Noting $P_{a\to b}$ the transition matrix transforming a vector $\vec{V_a}$ expressed in R_a into its representation $\vec{V_b}$ in R_b, we have:

$$\vec{V_b} = P_{a\to b}.\vec{V_a}$$

Thus, to express the geometry of said hand (natively defined in R_h) in camera space (hence in R_c), the following composition of referential changes[13] shall be applied:

$$P_{h\to c} = P_{s\to c}.P_{a\to s}.P_{f\to a}.P_{h\to f}.$$

[13]Thus transformation matrices, knowing that the product of such matrices is in turn a transformation matrix.

So a whole series of transformations can be done by applying a single matrix - whose coordinates are determined now.

Computing Transition Matrices For that, let's consider an homogeneous 4x4 matrix is in the form of:

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It can be interpreted as a matrix comprising two blocks of interest, R and $\vec{T}$:

$$M = P_{1\to 2} = \begin{bmatrix} R & \vec{T} \\ 0 & 1 \end{bmatrix}$$

with:

- R, which accounts for a 3D rotation submatrix:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

- $\vec{T}$, which accounts for a 3D translation vector:

$$\vec{T} = \begin{bmatrix} t1 \\ t2 \\ t3 \end{bmatrix}$$

Applying a (4x4 homogeneous) point $P = \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$ to M yields $P' = M.P$ where P' corresponds to P once it has been (1) rotated by R and then (2) translated by $\vec{T}$ (order matters).

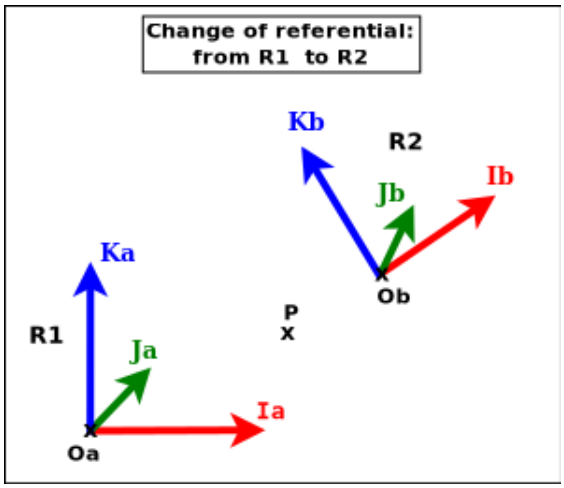
Let's consider now:

- two referentials (defined as orthonormal bases), R_1 and R_2; R_2 may for example be defined relatively to R_1; for a given point or vector U, U_1 will designate its coordinates in R_1 (and U_2 its coordinates in R_2)
- $P_{2\to 1}$ the (homogeneous 4x4) transition matrix from R_2 to R_1, specified first by blocks then by coordinates as:

$$P_{2\to 1} = \begin{bmatrix} R & \vec{T} \\ 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- any (4D) point P, whose coordinates are P_1 in R_1, and P_2 in R_2



The objective is to determine $P_{2\to1}$, i.e. R and $\vec{T}$.

By definition of a transition matrix, for any point P, we have: $P_1 = P_{2\to1}.P_2 \qquad (1)$

Let's study $P_{2\to1}$ by first choosing a point P equal to the origin of R_2 (shown as `Ob` in the figure).

By design, in homogeneous coordinates, $P_2 = Ob_2 = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{Bmatrix}$ and applying it on (1) gives us: $P_1 = Ob_1 = \begin{Bmatrix} t1 \\ t2 \\ t3 \\ 1 \end{Bmatrix}$.

So if $Ob_1 = \begin{Bmatrix} XOb_1 \\ YOb_1 \\ ZOb_1 \\ 1 \end{Bmatrix}$, we have: $\vec{T} = \vec{T_{2\to1}} = \begin{bmatrix} XOb_1 \\ YOb_1 \\ ZOb_1 \end{bmatrix}$.

Let's now determine the r_{xy} coordinates.

Let $R_{2\to1}$ be the (3x3) rotation matrix transforming any vector expressed in R_2 in its representation in R_1: for any (3D) vector $\vec{V}$, we have $\vec{V_1} = R_{2\to1}.\vec{V_2} \qquad (2)$

(we are dealing with vectors, not points, hence the origins are not involved here).

By choosing $\vec{V}$ equal to the $\vec{Ib}$ (abscissa) axis of R_2 (shown as `Ib` in the figure), we have $\vec{Ib_1} = R_{2\to1}.\vec{Ib_2}$

Knowing that by design $\vec{Ib_2} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, (2) gives us:

$$\vec{Ib_1} = \begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \end{bmatrix} = \begin{bmatrix} XIb_1 \\ YIb_1 \\ ZIb_1 \end{bmatrix}$$

So the first column of the R matrix is $I\vec{b}_1$, i.e. the first axis of R_2 as expressed in R_1.

Using in the same way the two other axes of R_2 (shown as `Jb` and `Kb` in the figure), we see that:

$$R = R_{2\to1} = \begin{bmatrix} XIb_1 & XJb_1 & XKb_1 \\ YIb_1 & YJb_1 & YKb_1 \\ ZIb_1 & ZJb_1 & ZKb_1 \end{bmatrix}$$

So the transition matrix from R_2 to R_1 is:

$$P_{2\to1} = \begin{bmatrix} R_{2\to1} & \vec{T_{2\to1}} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} XIb_1 & XJb_1 & XKb_1 & XOb_1 \\ YIb_1 & YJb_1 & YKb_1 & YOb_1 \\ ZIb_1 & ZJb_1 & ZKb_1 & ZOb_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

- $R_{2\to1}$ is the 3x3 rotation matrix converting vectors of R_2 in R_1, i.e. whose columns are the axes of R_2 expressed in R_1
- $\vec{T_{2\to1}} = Ob_1$ is the 3D vector of the coordinates of the origin of R_2 as expressed in R_1

This also corresponds to a matrix obtained by describing the R_2 referential in R_1, by listing first the three (4D) vector axes of R_2 then its (4D) origin, i.e. $P_{2\to1} = \left[I\vec{b}_1 \quad J\vec{b}_1 \quad K\vec{b}_1 \quad Ob_1\right]$.

Often, transformations have to be used both ways, like in the case of a scene-to-camera transformation; as a consequence, transition matrices may have to be inversed, knowing that $(P_{2\to1})^{-1} = P_{1\to2}$ (since by definition $P_{2\to1}.P_{1\to2} = Id$).

An option to determine $P_{1\to2}$ from $P_{2\to1}$ could be to compute its inverse directly, as $P_{1\to2} = (P_{2\to1})^{-1}$, yet $P_{1\to2}$ may be determined in a simpler manner.

Indeed, for a given point P, whose representation is P_1 in R_1 and P_2 in R_2, we obtain $P_1 = P_{2\to1}.P_2$ by - through the way (4x4) matrices are multiplied - first applying a (3x3) rotation Rot_3 to P_2 and then a (3D) translation T_r: $P_1 = Rot_3.P_2 + T_r$ (in 3D; thus leaving out any fourth homogeneous coordinate); therefore $P_2 = (Rot_3)^{-1}.(P_1 - T_r)$. Knowing that the inverse of an orthogonal matrix is its transpose, and that rotation matrices are orthogonal, $(Rot_3)^{-1} = (Rot_3)^{\top}$, and thus $P_2 = (Rot_3)^{\top}.(P_1 - T_r) = (Rot_3)^{\top}.P_1 - (Rot_3)^{\top}.T_r)$. So if:

$$P_{2\to1} = \begin{bmatrix} R_{2\to1} & \vec{T_{2\to1}} \\ 0 & 1 \end{bmatrix}$$

then:

$$P_{1\to2} = \begin{bmatrix} (R_{2\to1})^{\top} & -(R_{2\to1})^{\top}.\vec{T_{2\to1}} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} XIb_1 & YIb_1 & ZIb_1 & -(XIb_1.XOb_1 + YIb_1.YOb_1 + ZIb_1.Z \\ XJb_1 & YJb_1 & ZJb_1 & -(XJb_1.XOb_1 + YJb_1.YOb_1 + ZJb_1.Z \\ XKb_1 & YKb_1 & ZKb_1 & -(XKb_1.XOb_1 + YKb_1.YOb_1 + ZKb_1.Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note

So, in a nutshell, the transition matrix from a referential R_α to a referential R_β is:

$$P_{\alpha\to\beta} = \begin{bmatrix} Rot_{\alpha\to\beta} & \vec{Tr_{\alpha\to\beta}} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} XIb_\beta & XJb_\beta & XKb_\beta & XOb_\beta \\ YIb_\beta & YJb_\beta & YKb_\beta & YOb_\beta \\ ZIb_\beta & ZJb_\beta & ZKb_\beta & ZOb_\beta \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

- $Rot_{\alpha\to\beta}$ is the 3x3 rotation matrix converting vectors of R_α in R_β, i.e. whose columns are the axes of R_α expressed in R_β
- $\vec{Tr_{\alpha\to\beta}} = Ob_\beta$ is the 3D vector of the coordinates of the origin of R_α as expressed in R_β

This also corresponds to a matrix obtained by describing the R_α referential in R_β, by listing first the three (4D) vector axes of R_α then its (4D) origin, i.e. $P_{\alpha\to\beta} = \left[\vec{Ib_\beta} \quad \vec{Jb_\beta} \quad \vec{Kb_\beta} \quad Ob_\beta\right]$.

Its reciprocal (inverse transformation) is then:

$$P_{\beta\to\alpha} = \begin{bmatrix} (Rot_{\alpha\to\beta})^\top & -(Rot_{\alpha\to\beta})^\top.\vec{Tr_{\alpha\to\beta}} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} XIb_\beta & YIb_\beta & ZIb_\beta & -(XIb_\beta.XOb_\beta + YIb_\beta.YO \\ XJb_\beta & YJb_\beta & ZJb_\beta & -(XJb_\beta.XOb_\beta + YJb_\beta.YO \\ XKb_\beta & YKb_\beta & ZKb_\beta & -(XKb_\beta.XOb_\beta + YKb_\beta.YO \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As a result, from the definition of a tree of referentials, we are able to compute the transition matrix transforming the representation of a vector expressed in any of them to its representation in any of the other referentials.

A special case of interest is, for the sake of rendering, to transform, through that tree, a local referential in which a geometry is defined into the one of the camera, defining where it is positioned and aimed[14]; in OpenGL parlance, this corresponds to the *model-view* matrix (for "modelling and viewing transformations") that we designate here as M_{mv} and which corresponds to $P_{local\to camera}$.

Taking into account the last rendering step, the *projection* (comprising clipping, projection division and viewport transformation), which can be implemented as well thanks to a 4x4 (non-invertible) matrix designated here as M_p, we see that a single combined overall matrix $M_o = M_p.M_{mv}$ is sufficient[15] to convey in one go all transformations that shall be applied to a given geometry for its rendering.

[14] `gluLookAt` can define such a viewing transformation matrix, when given (1) the position of the camera, (2) a point at which it shall look, and (3) a vector specifying its up direction (i.e. where is the upward direction for the camera - as otherwise all directions orthogonal to its line of sight defined by (1) and (2) could be chosen).

[15] In practice, for more flexibility, in OpenGL the management of the viewport, of the projection and of the model-view transformations is done separately (for example, respectively, with: `glViewport`, `glMatrixMode(GL_MODEL-VIEW)` and `glMatrixMode(GL_PROJECTION)`; so there is a matrix stack corresponding to `GL_MODEL-VIEW` and another one to `GL_PROJECTION`).

Main Matrices

These matrices account for the main processing steps of a rendering.

Three types of referentials can be considered:

- **world** referential: the absolute, overall referential where 3D scenes are to be assembled
- **local** referential: a referential in which a given model is defined (generally located at its origin)
- **camera** referential: a referential where a camera is at the origin, looking down on the negative Z axis

The **clip space** can also be considered; this is the post-projection space, where the view frustum is transformed into a cube, centered in the origin, and going from -1 to 1 in every axis.

The transformations between referentials can be represented by 4×4 transition matrices:

- **model matrix** (M_M): to transform from local to world referential
- **view matrix** (M_V): to transform from world to camera referential
- **projection matrix** (M_P): to transform from camera referential to clip space

Finally, two composite matrices are especially useful (note the aforementioned reverse multiplication order) and are typically passed through uniform variables in shaders:

- ModelView: $M_{MV} = M_V.M_M$
- ModelViewProj: $M_{MVP} = M_P.M_{MV} = M_P.M_V.M_M$

Shaders

They are in-depth covered in the Khronos wiki.

A Programmable Pipeline Shaders are the basic rendering building blocks of applications using modern OpenGL (ex: 3.x/4.0).

Such an application will indeed program its own shaders, instead of calling functions like `glBegin()/glEnd()`, as it was done with OpenGL 1.x-2.x and its fixed-pipeline immediate mode.

This mode of operation, albeit more complex, offers more control and enables increased performances.

Parallelism in the Pipeline The key is to write programs that can be executed in a *Single Instruction, Multiple Data* (SIMD) setting, in order to take advantage of the vectorization typically supported by GPUs.

A goal is to avoid conditional branching based on values that may differ from a shader invocation to another (see this explanation).

If having to take into account two dynamically-uniform (i.e. non-statically predictable, yet having the same value for very shader invocation within that group) branches performing simple computations, it is likely that the compiler will generate code evaluating both expressions, until dropping the result of the one finally not happening.

Six Types of GLSL Shaders Shaders are written in the **GLSL** language, i.e. the OpenGL Shading Language.

They are portions of C-like code that can be inserted in the rendering pipeline implemented by the OpenGL driver of a GPU card. Six different kinds of shaders can be defined, depending on the processing step that they implement and on their purpose: vertex, tessellation for control or for evaluation, geometry, fragment or compute shaders.

Except this last type (compute shader), all types are mostly dedicated to *rendering*. If wanting to perform on one's GPU more general-purpose processing, OpenCL shall be preferred to GLSL.

Runtime Build Shaders are compiled at (application) runtime[16] (to target exactly the actual hardware), then linked and attached to a separate program running on the GPU. This is fairly low-level, black-box direct programming, in sharp contrast with the reliance on APIs that used to be the norm with OpenGL 1.x.

Yet offline compilers exist as well, as well as debuggers (like the NVIDIA NsightShader Debugger).

Implementing a Shader A shader is quite similar to a C program, yet based on a specific, core language that enables the definition of relevant data types and functions.

Data types are usually based on elementary types (`float`, `double`, `bool`, `int` and `uint`), and composed in larger structures, like `{vec,mat}{2,3,4}`, `mat2x3`, arrays and structures, possibly `const`; see this page for further details.

Similarly, control flow statements and (non-recursive) functions can be used; every shader must have a `main` function, and can define other auxiliary functions as well, in a way similar to a C program. Function parameters may have the `in` (which is the default), `out` or `inout` qualifiers specified. Additionally a function may return a result, thanks to `return`.

So, regarding output, for example a fragment shader must return the color that it computed; `out vec3 my_color;` declares this; and the shader code may be as simple as returning a constant color in all cases, like in:

```
#version 330 core
out vec3 my_color;

void main()
{
   // Same color returned for all fragments:
   my_color = vec3(0.05, 0.2, 0.67);
```

[16] So each shader is built each time the application is started, and the operation may fail (ex: with `0(40) : error C1503: undefined variable "foobar"`).

```
}
```

Communicating with Shaders Of course the application must have a way of supplying information to its shaders (the other way round does not really happen, except for compute shaders), and a given shader must be able to pass information to (only) any next shader in the pipeline.

Two options exist for shaders to have inputs and outputs, from/to the CPU and/or other shaders:

- basically each shader is fed with a stream of vertices[17] with associated data, named **vertex attributes**; they are either *user-defined* or *built-in* (each type of shader having its own set of built-in input attributes)
- global, read-only data can also be defined, as **uniform** variables

These communication options are discussed more in-depth next.

Vertex Attributes **Defining Attributes**

A vertex attribute, whether user-defined or built-in, may store any kind of data - notably positions, texture coordinates and normals.

Either a given attribute is a single, *standalone* one (then a unique value will be read and will apply to all vertices), or is *per-vertex*, in which case it is read from a buffer, each element of it being bound accordingly when its associated vertex is processed by the shader. Such arrays are either used in-order, or according to any indices defined (then themselves defined thanks to an array as well)[18].

Said differently, for each attribute used by a shader, either a single value or an array thereof must be specified.

Referencing Attributes

In order that attributes can be referenced from at least one shader, they must be matched:

- by *name*: then they must bear exactly the same name in the main program and in the shaders using them, knowing that any name beginning with `gl_` is reserved
- or by *location*: their common location is specified, and a per-shader variable name is associated - which is more flexible
- or by *block*, like for the uniform variables, discussed below

[17] Then the user-defined *primitives*, applied later in the pipeline, will allow OpenGL to interpret such a series of vertices in terms of a sequence of triangles, or points, or lines, etc.

[18] This is the preferred method, as it prevents vertex duplication, and allows to process each of them once: there is a vertex cache that stores the outputs of the last processed vertices, so that if a vertex is mentioned multiple times (ex: being included in a triangle fan or strip), the corresponding output may be directly re-used (provided it is still in the cache) instead of having to be computed again.

In the two last cases, the layout of variables must match on either side (ex: main program/shader); for example, with `"layout(location = 0) in vec3 input_vertex;"` in its code, a vertex shader will expect a (single) vector of 3 (floating-point) coordinates (`vec3`) to be found at index 0 (`location = 0`) as input (`in`); the application will need to specify a corresponding *Vertex Buffer Object* (VBO) for that.

So that they can be fetched for a given vertex, attributes have to be appropriately located in buffers. For each attribute, either the developer defines, prior to linking, a specific location (as an index starting at zero) with `glBindAttribLocation`, or he lets OpenGL choose it, and queries it afterwards, with `glGetAttribLocation`; refer to this page for further details.

If a given program is linked with two shaders, a vertex one and a fragment one, the former one will probably have to pass its outputs as inputs of the latter one; this requires as many variables defined on either side, with relevant out/in specifications, and a matching name and type; for example the vertex shader may declare `out vec3 my_Color;` whereas the fragment shader will declare `in vec3 my_Color;`.

Providing Attribute Data : VAO and VBO

Vertex data is provided thanks to a (single current) Vertex Array Object (VAO).

A VAO references (rather than storing directly) the format of the vertex data, as well as the buffers (VBOs, see below) holding that data.

A vertex attribute is identified by a number (in `[0;GL_MAX_VERTEX_ATTRIBS-1]`), and by default is accessed as a single value (as opposed to as an array).

A Vertex Buffer Object (VBO) is a data array, typically referenced by a VAO. A VBO defines its internal structure and where the corresponding data can be found.

So, in practice, each homogeneous chunk of data to be sent from the CPU-space to the GPU-space (vertices, normals, colors) is stored in an array corresponding to a *Vertex Buffer Object* (VBO), itself stored in a *Vertex Array Object* (VAO). A VAO may gather vertex data and colour data in separate VBOs, and store them on the graphics card for any later use (as opposed to streaming vertices through to the graphics card when they become needed). A VAO is only meant to hold one (VBO) array of vertices, each other VBO being used then for per-vertex attributes.

Uniform Variables are Read-Only and Global Instead of relying on attributes, an alternate way of passing information, provided that it may change relatively infrequently, is to use *uniform variables*, which behave, for a shader and in the course of a draw call, as read-only, constant global variables for all vertices (hence their *uniform* naming). Any shader can access every uniform variable (they are globzl), as long as it declares that variable.

Examples of uniform variables could be the position of a light, transformation matrices, fog settings, variables such as gravity and speed, etc.

Uniform variables may be defined individually, or be grouped in named blocks, for a more effective data setup (to share uniform variables between programs) and transfer from the application to the shader (setting multiple values at once).

Uniform variables are declared at the program-level (as opposed to a per-vertex level) thanks to:

- the `uniform` qualifier on the shader-side, like in `uniform mat4 MyMatrix;`
- a `glGetUniformLocation` call on the application-side, to create a location associated to a name (ex: `"MyMatrix"`), and to associate it to a given value, like in:

```
mat4 someMatrix = [...];

GLuint location = glGetUniformLocation(programId, "MyMatrix");

if (location >= 0)
{
   // Defining a single matrix (1), not to transpose (GL_FALSE):
   glUniformMatrix4fv(location, 1, GL_FALSE, &someMatrix[0][0]);
   [...]
```

Individual variables may be used as uniform, as well as arrays and structs.

From the point of view of a shader, these named input variables may be initialised when declared, but then are read-only; otherwise the application may choose to set them.

Built-in Variables are Defined by Each Shader Type Finally, depending on the type of a shader, some predefined, built-in variables ("intrinsic attributes") may be set; they are specified here.

For example, for a vertex shader, following output variables are predefined:

- `vec4 gl_Position` corresponding to the clip-space (homogeneous) position of the output vertex
- `float gl_PointSize`
- `float gl_ClipDistance[]`

Examples of Shaders See the ones of Wings3D (in GLSL `"1.2"` apparently, presumably for maximum backward compatibility; note that some elements, with the `*.cl` extension, are OpenCL ones), or these ones.

Managing Spatial Transformations Modern OpenGL (and GLU) implementations basically dropped the direct matrix support (the so-called immediate mode does not exist anymore, except in a compatibility context). So no more calls to `glTranslate`, `glRotate`, `glLoadIdentity` or `gluPerspective` shall be done; now the application has to compute such matrices (for model, view, texture, normal, projection, etc.) by itself (on the CPU), as inputs to its GLSL shaders.

For that, applications may use dedicated, separate libraries, such as, in C/C++, GLM, i.e. OpenGL Mathematics (Myriad's linear support aims to provide, in Erlang, a relevant subset of these operations).

The matrices that correspond to the transformations to be applied are then typically shared with the shaders thanks to uniform variables.

This is especially the case for the vertex shader, in charge of transforming coordinates expressed in a local referential into screen coordinates.

More precisely, the modeling (object-space to absolute, world-space), viewing (world-space to camera-space) and projection transformation (camera-space to clip-space) are applied in the vertex shader, whereas the final perspective division and the viewport transformation are applied in the fixed-function stage after the vertex shader.

So a vertex shader is usually given two 4x4 homogeneous, uniform matrices:

- a modelview matrix, combining modeling and viewing, to transform object-space to camera-space in one go
- a projection matrix

More Advanced Topics

Shadows Determining the shadow of an arbitrary object on an arbitrary plane (representing typically the ground - or other objects) from an arbitrary light source (possibly at infinity) corresponds to performing a specific **projection**. For that, a relevant 4x4 (based on homogeneous coordinates) matrix (singular, i.e. non-invertible matrix) can be defined.

This matrix can be multiplied with the top of the model-view matrix stack, before drawing the object of interest in the shadow color (a shade of black generally).

Refer to this page for more information.

Sources of Information

The reference pages for the various versions of OpenGL are available on the Khronos official OpenGL Registry.

Two very well-written books, strongly recommended, that are still relevant for 3D graphics despite their old age (circa 1996; for OpenGL 1.1):

- *The Official Guide to Learning OpenGL*: the OpenGL Red book
- *The OpenGL Reference Manual*: the OpenGL Blue book

More modern tutorials (applying to OpenGL 3.3 and later) are:

- Opengl-tutorial
- Learn OpenGL
- regarding GLSL shaders: lighthouse3d

Other elements of interest:

- the OpenGL 3.3 Specification (Core Profile) (the 425-page reference PDF)
- FAQ for OpenGL and GLUT

- the (archived) OpenGL FAQ and Troubleshooting Guide, containing much valuable information, including regarding transformations
- About OpenGL Performance
- in French: Introduction à OpenGL et GLUT, by Nicolas Roussel
- any textbook on linear algebra

Operating System Support for 3D

Benefiting from a proper 2D/3D hardware acceleration on GNU/Linux is unfortunately not always straightforward, and sometimes brittle.

The very first step is to **update's one's video drivers to their latest, official stable version** according to your OS/distribution of choice (even if it implies using closed-source binaries...) and check that they are in use (probably reboot then).

Testing

First, one may check whether such acceleration is already available by running, from the command-line and as the current, non-priviledged user, the `glxinfo` executable (to be obtained on Arch Linux thanks to the `mesa-utils` package), and hope to see, among the many displayed lines, `direct rendering: Yes`.

One may also run our display-opengl-information.sh script to report relevant information.

A final validation might be to run the `glxgears` executable (still obtained through the `mesa-utils` package), and to ensure that a window appears, showing three gears properly rotating.

Troubleshooting

If it is not the case (no direct rendering, or a GLX error being returned - typically involving any `X Error of failed request: BadValue` for a `X_GLXCreateNewContext`), one should investigate one's configuration (with `lspci | grep VGA`, `lsmod`, etc.), update one's video driver on par with the current kernel, reboot, sacrifice a chicken, etc.

If using a NVidia graphic card, consider reading this Arch Linux wiki page first.

In our case, installation could be done with `pacman -Sy nvidia nvidia-utils` but requested a reboot.

Despite package dependencies and a not-so-successful attempt of using DKMS in order to link kernel updates with graphic controller updates, too often a proper 3D support was lost, either from the boot or afterwards. Refer to our software update section for hints in order to secure the durable use of proper drivers.

Minor Topics

Camera Navigation Conventions

Multiple tools introduced conventions in order to navigate, with mouse and keyboard, in a 3D world.

We prefer the way Blender manages the observer viewpoint (current camera), as described here; notably, supposing a three-button mouse with a scrollwheel:

- `orbit the view around the currently selected object` (or `Tumble`) by holding the middle button down and moving the mouse
- `pan` (moving the view up, down, left and right) by holding down Shift and the middle button, and moving the mouse
- `zoom in/out` with the scrollwheel; a variation of it, `Dolly`, can be obtained by holding down Ctrl and the middle button, and moving the mouse

3D-Related Mini-Glossary

- **HDRP**: *High Definition Render Pipeline*, a high-fidelity scriptable render pipeline, made by Unity to target **modern** (Compute Shader compatible) platforms (so HDRP is the high-end counterpart of URP)
- **IK**: *Inverse Kinematics*, the **computation of intermediary joint parameters** so that the end of the kinematic chain is at a given position and orientation; typically, if one wants the hand of a character to grasp the top of a chair, IK is used in order to determine the parameters of the character's wrist, arm, elbow, etc. that may be retained so that the hand is ultimately correctly placed on the chair (more information)
- **Material**: controls the optical properties of an object, i.e. how a 3D object appears on the screen, that is: the color of each point of the object (generally thanks to multiple texture maps, like diffusion, normal, specular, glow, etc.) and how reflective or dull its surface appears; designates, with OpenGL, a set of coefficients that define **how the lighting model interacts with the surface**; in particular, ambient, diffuse, and specular coefficients for each color component (R,G,B) are defined and applied to a surface and effectively multiplied by the amount of light of each kind/color that strikes the surface; a final emmissivity coefficient is then added to each color component so that objects can also be light emitters
- **NURBS**: *Non-Uniform Rational B-Spline*, a mathematical model using basis splines (B-splines) that is commonly used in computer graphics for representing **curves and surfaces**, whose shape is determined by control points (more information)
- **PBR**: *Physically-Based Rendering* designates approaches to render images in a way that **models the flow of light in the real world**, for example thanks to photogrammetry; many PBR pipelines aim to achieve photorealism; in practice they often rely on the **micro-facet theory**, with specific materials (generally based on texture maps) and shaders (is also called PBS, for *Physically-Based Shading*); PBR is slowly becoming the standard for all materials
- **PSD**: *Photoshop Document*, a proprietary format for graphics with layers, masks, etc. used by Adobe Photoshop (a commercial counterpart to Gimp, Krita, etc.) often used to store textures that may still be edited as templates by the user - provided they are using Photoshop as well;

however, at least to some extent, Gimp is able to edit PSD files and Krita too

- **Rigging** (or *Skeletal Animation*) consists in **controlling the deformation of a mesh** (a.k.a. a *skin*, the surface of a body) of an articulated object (typically a character) **based on a virtual inner armature** (a hierarchical set of interconnected parts, called *bones*, and collectively forming the skeleton or *rig*) in order to animate that mesh (more information)
- **Textures**: bitmaps (images) used to **skin 3D objects**, by defining the color of each point on the surface of the object in terms of texture coordinates; besides such 2D textures, 1D, 3D and 4D ones exist
- **Texture Atlas**: a texture (an image) containing a **set of separate, elementary graphic elements**, meant to be extracted based on texture coordinates, akin to a sprite sheet; doing so is useful to reduce the overhead that would be induced by the management of many smaller textures (more information)
- **URP**: *Universal Render Pipeline*, a prebuilt scriptable render pipeline, made by Unity, which implements workflows across a range of platforms, from mobile to high-end consoles and PC (in practice URP is the low-end counterpart of HDRP)

See also the Wikipedia's glossary of computer graphics.

Network Management

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Tuesday, August 30, 2022

Investigating Network Issues

Tools like `ping`, `traceroute`, `drill`, `arp`, etc. are invaluable.

Use ip-scan.sh to scans all IPs with any specified prefix, and ip-examine.sh to collect information about a given IP.

Use monitor-network.sh to investigate unstable connections.

Firewall Management

On GNU/Linux, some level of knowledge about `iptables` is useful, notably if exposing a computer to the Internet; note though that it is to be superseded by nftables.

One should read first the very clear Arch wiki section about iptables basic concepts.

A general rule that we retain, especially for an Internet gateway, is to drop all packets by default, and then only to accept the expected ones explicitly and carefully.

Configuration of a Gateway to the Internet

Our iptables.rules-Gateway.sh script sets up an iptables configuration with various services that can be enabled (ex: for masquerading, IPTV, different kinds of servers) as an example that we hope is secure enough[19].

This script expects a settings file to be available as `/etc/iptables.settings-Gateway.sh` (this file is meant to be sourced, not executed).

An example thereof:

```
# Local firewall settings.
#
# Meant to be sourced by the iptables.rules-Gateway.sh script.

# Where firewall-related outputs will be written:
log_file=/root/.last-gateway-firewall-activation

# Local (LAN) interface, the one we trust:
#lan_if=eth1
lan_if=enp2s0

# Internet (WAN) interface, the one we distrust:

# For PPP ADSL connections:
```

[19] Please email us if you found otherwise! Refer to the top of this document for that.

```
#net_if=ppp0

# For direct connection to a set-top (telecom) box from your provider:
#net_if=eth0
net_if=enp4s0

ban_file="/etc/ban-rules.iptables"

# As the IPs banned through the ban file above are quite minimal:
use_ban_rules="true"
#use_ban_rules="false"

# IP of a test client (to avoid too many logs, selecting only related events):
#test_client_ip="xxx"

# Enabled input TCP port range for traffic from LAN to gateway:
enable_unfiltered_tcp_range="true"

# TCP unfiltered window (ex: for passive FTP and BEAM port ranges):
tcp_unfiltered_low_port=50000
tcp_unfiltered_high_port=55000

# Tells whether IPTV (TV on the Internet thanks to a box) should be allowed:
enable_iptv=false

# Tells whether a SMTP server can be used:
enable_smtp=false

# Typically a set-top box from one's ISP (defined as a possibly log match
# criteria):

# Classical example:
telecom_box="192.168.0.254"

# DHT subsection, for P2P exchanges:
# More infos: https://github.com/rakshasa/rtorrent/wiki/Using-DHT

dht_udp_port=7881

#use_dht="true"
use_dht="false"

# One may use a non-standard port:
#ssh_port=22
ssh_port=22320

smtp_port=25

# SMTPS is obsolete:
smtp_secure_port=465
```

```
# STARTTLS over SMTP is the proper way of securing SMTP:
msa_port=587

pop3_port=110

# POP3S:
pop3_secure_port=995

imap_port=143
imap_secure_port=993
```

A script to configure iptables is best integrated to systemd, see the iptables.rules-Gateway.service file for that (typically to be placed in `/etc/systemd/system`). Then one may test with:

```
$ systemctl start iptables.rules-Gateway.service
```

and enable it for good with:

```
$ systemctl enable iptables.rules-Gateway.service
```

Note that often these scripts are setup remotely, while being connected thanks to SSH from another host. Care must be taken in order not to lock oneself out of the target server, notably when updating rules (this happens quite easily). We advise to prefer the `restart` option of our iptables script in order to reduce the risk of "bricking" one's server.

Firewall-related Troubleshooting

Use iptables-inspect.sh to list the currently-used firewall rules for the chains of the main tables. Like `iptables -nL --line-numbers`, it displays the number of each rule of a given chain, which allows to add/remove rules more easily, like in:

```
# Deletes the first rule of the FORWARD chain (of the 'filter' table):
# (note that all the next rules will bear a decremented number afterwards!)
$ iptables -D FORWARD 1
```

Setting environment variables (either through files such as `/etc/iptables.settings-Gateway.sh` or directly in the shell) is less error-prone; ex:

```
[...]
$ lan_if=enp2s0
$ net_if=enp4s0
$ iptables -I FORWARD -i ${lan_if} -o ${net_if} -d ${telecom_box} -j LOG
$ journalctl -kf --grep="IN=.*OUT=.*" | grep -v "SRC=${telecom_box}"
```

To further match packets, one may specify log prefixes, like in:

```
$ iptables -A INPUT -i lan.foobar -j LOG --log-prefix "[VLAN INP FOO]"
```

Note that the `LOG` target does not intercept a packet, which thus continues to flow in the next rule(s). so log targets are better defined as first rules (and thus could be inserted lastly).

As a reminder, for a given table (`filter` by default), rules may be:

- appended *at the end* of the selected chain with `-A`
- inserted either at the *beginning* of the selected chain with `-I`, or at its position N with `-I N`

See also the iptables section in the Arch wiki.

Network Troubleshooting

A few pieces of advice/information:

- be familiar with `ip link`, `ip addr` and `ip route` (generally used in that order), and `tcpdump` for the worst cases
- to review *host-local* open ports and sockets, use `ss` (for *socket statistics*, replacement of `netstat`; ex: `ss --inet --listening -p`) or `nmap` for remote testing
- nowadays, many devices change their MAC address regularly, like smartphones do
- one may rely on `netctl`, and create as many profiles as found useful
- regularly inspect network-related messages (ex: with `journalctl -kf`) to detect anomalies such as `IPv4: martian source 192.168.0.49`
- interfaces may be associated to any number of IP addresses, this may create surprises
- when a network does not work properly, always consider that this device may be faulty, that cables may malfunction, and that power supplies may be culprits
- having smart switches may help a lot, to better control one's network (ex: disabling ports, checking statuses, isolating sections, etc.)
- beware to DHCP server(s) being left unnoticed; various devices may use them to get a random address and become difficult to spot
- netmasks shall not be neglected, for example in routes:

```
$ ip route add 192.168.0.0/16 dev enp4s0 scope link
$ ip route
default via 192.168.0.254 dev enp4s0 proto dhcp src 192.168.0.1 metric 1002
10.0.0.0/8 dev enp2s0 proto kernel scope link src 10.0.0.1
192.168.0.0/16 dev enp4s0 scope link
```

Here for example, in `192.168.0.0/16`, `16` corresponds to the length of *the network prefix*; the next 16 bits are left to designate hosts, whose addresses therefore range in `192.168.[0..254].[1..254]`. So `192.168.0.0/16` includes the `192.168.27.0/24` network, whereas `192.168.0.0/24` would not.

- go for VLAN only when having reached a first level of correct operation; note that some devices (ex: non-manageable switches) are not able to handle VLAN-tagged packets and may reject or overwrite this information
- in some cases, hard reboots / returns to factory settings will fix inexplicable situations; updating to latest firmware may help too (network appliances *do* have bugs as well!)
- secure spare parts (if possible all cables, fibers, devices, power supply, etc. shall exist at least in two copies, tested just after purchased): when the one in operation will fail, the outage will be quickly solved by switching element; the troubleshooting will be easier as well: replace the whole set of equipment, check that everything works again, and try to progress by dichotomy (change half of the elements, and check whether everything remains functional)
- purchase only equipment of quality, and treat it gently (ex: use an Uninterruptible Power Supply providing good-quality current)
- take notes about the operations that are performed, the detected issues and the current configuration, and put the whole in VCS
- check temperature, ventilation and prevent dust accumulation
- consider monitoring temperatures, fans, availability, performances

See Also

- Ceylan-Hull's section about scripts for network management and for firewall configuration
- A bit of Cybersecurity

A Bit of Cybersecurity

Organisation: Copyright (C) 2021-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Friday, April 8, 2022

Pointers to various Security Topics

A goal here is to favor cryptographic privacy and authentication for data communication.

More precisely:

- for **data storage** (be it a USB key or a SSD disk), it may translate to partition encryption, typically with LUKS2 and cryptsetup
- **individual files** may be encrypted/decrypted with the help of appropriate scripts; see also Ceylan-Myriad's support for additional basic, old-school ciphering
- for the **management of credentials** such as passwords, some Ceylan-Hull scripts may be of help, including for the generation of proper passwords or for the locking of screens
- regarding **network**, each host may be protected by a relevant firewall configuration, opened ports may be checked, etc.; see also our section for firewall management
- for **webservers**, it relates to use the HTTPS protocol with proper X.509 security certificates for TLS-secured exchanges, possibly thanks to Ceylan-LEEC
- for **emails**, see the next section about OpenPGP

Securing thanks to OpenPGP

Purpose

Albeit such a securing scheme may apply to at least most of the digital exchanges, in practice it is mainly used in the context of email security.

In the general case, sending an email will end up having its content stored at least on:

- your disk
- a disk of one of the servers of your Internet provider
- a disk of a server of the provider of the recipient
- the recipient's host

Possibly with intermediate organisations between the endpoint ones, possibly stored on several locations per organisation - possibly times the number of specified recipients.

Moreover many countries require by law that emails are stored by Internet providers durably (often at least for one year) - not to mention the large-scale data harvesting that many countries perform, officially or not, with their own measures, on their own territory or on the one of others.

That's a rather large number of copies for one's private correspondence - to the point that emails sent in clear text could be mostly considered as public. Not to mention that they could also be altered in the process, at some point(s) in the chain.

Encrypting and signing are solutions to restore some privacy and safety - yours, but also the ones of the persons with whom you happen to correspond.

Technical Solution

It is currently best done thanks to the OpenPGP open standard for encrypting, signing and decrypting data and communications.

GnuPG (*GNU Privacy Guard*) is a complete and free implementation of it (we suppose here that at least its 2.2.* version is used).

The corresponding command-line executable, `gpg`, can be installed on Arch Linux with: `pacman -Sy gnupg`.

Obtaining One's Keys The first step is to generate locally one's key pair, knowing that each public key is bound to a username or an e-mail address (which is our preference; having one's domain name allows to create any number of them).

A nice feature of this cryptographic scheme is that one may issue any number of keys in full autonomy and with neither consequences nor cost. So as many key pairs as notions of "unrelated identities" may be freely created.

Several settings can be chosen when generating a key, and logically the strongest keys are preferred. Yet uncommon/too recent generation algorithms and/or higher key lengths may not be supported by the various tools[20], so applying the default settings retained by gpg, or similar ones yet a bit stronger (ex: at the time of this writing, November 2021, RSA 4096 bits rather than 3072 bits) is probably the way to go (it can already be deemed safe, and will be widely supported); so the generation may be best triggered simply thanks to:

```
# For current defaults:
$ gpg --gen-key

# Or, for more control:
$ gpg --full-gen-key
```

[20] With "cutting-edge" settings, some tools (like Thunderbird) on your side and/or the email clients of your recipients may be unable to make use of the resulting keys, and may fail to report clearly that they actually do not support this algorithm or its parametrisation. So one may consider sticking to the reasonable gpg defaults.

If preferring rather paranoid settings, presumably for an extra security/durability, one can select ECC (for Elliptic-curve cryptography), with the `Sign`, `Certify` and `Authenticate` capabilities enabled (even if authentification is not used by many common protocols), and opt for the `Brainpool P-512` curve through:

```
$ gpg --full-gen-key --expert
```

In all cases, one may enter `1y` to set the initial validity duration of the generated key to one year, and already plan in one's agenda, a dozen days before the end of its validity, its renewal.

Then one may enter one's selected identity (ex: for `Real name`, one may enter `James Bond`), one's email address of interest (ex: `james.bond@mi6.org`) and possibly:

- either no specific comment (they are not normalised anyway)
- or one pointing to an authoritative source against which the public key may be verified (such as: "This public key can be verified against its reference in https://mi6.org/james-bond.pub" - provided of course such a file is to exist)

The requested passphrase only consists on a last-resort protection of the generated private key (that you should *never* transmit to *anyone*), in order to avoid that anyone accessing this file on your computer becomes directly able to fully impersonate this identity.

The operation generates a public/private key pair, and also an associated emergency revocation certificate, so that you can invalidate it at any time and for any reason:

```
gpg: key 9A60ADA4E151B8B5 marked as ultimately trusted
gpg: directory '/home/james/.gnupg/openpgp-revocs.d' created
gpg: revocation certificate stored as '/home/james/.gnupg/openpgp-revocs.d/C3987680AD9
public and secret key created and signed.

pub   brainpoolP512r1 2021-11-26 [SCA] [expires: 2022-11-26]
      C3987680AD9B79FDC6B7D25C9D60ADA5E115A8B5
uid   James Bond <james.bond@mi6.org>
```

Here `C3987680AD9B79FDC6B7D25C9D60ADA5E115A8B5` is the full fingerprint of the public key; it could be shortened to its 8, if not 4, last characters (long/short ID), yet it would expose to the forging of intentionally-colliding keys, so one should only designate a key based on its full fingerprint, and forget unsafe abbreviations.

The public key can be freely shared, whereas the private one and the revocation certificate must be equally well protected (preferably in different places).

The only well-known threats to these keys are either a flaw (intentional loophole or accidental weakness) in the cryptographic algorithms on which they rely, or the advent of major research progresses such as quantum computing. Yet it still remains possible for one to "upgrade" one's key with newer algorithms (a new key superseding an older one that is to be revoked afterwards), so as always it will be a never-ending struggle between the spear and the shield, i.e. attack and defense.

As signing and encrypting correspond to different use cases, having different keys for each may make sense. But instead of generating two unrelated keys, one shall create:

- first an infrequently-used, very-well protected (hence less accessible), signing-only "master" (primary) key of longer validity (one's actual identity)
- then at least two subkeys (deriving from the previous one, yet autonomous) may be of use:
  - one for everyday encrypting; a proper subkey has already been automatically created and used by gnupg
  - an extra one for everyday signing: such a subkey may be created with a sufficient lifespan so that past signatures can be durably verified

These "derived" subkeys are meant to change more frequently, to be able to be revoked independently, and thus are safer to expose in less secure systems.

Use `gpg --edit-key` and `addkey` in order to add a subkey to a key, and refer to this section to export the subkey.

See also these very relevant Debian guidelines for further information about subkey management.

Where are the keys, and how to backup them? The full gpg state is stored by default in its `~/.gnupg/` tree.

One may notably notice in it:

- the private keys, whose extension is `.key` and whose security is of course of paramount importance
- the revocation certificates, whose extension is `.rev`, in order to revoke one's corresponding key pair (as important as the related private key)
- certificate revocation lists, to consider that the corresponding certificates are valid yet shall *not* be trusted
- the sets of keys ("rings") containing the public keys that have been transmitted to you, gathered according to the level of trust that you dedicated to them

The public keys are usually given a `.pub` extension[21].

Even if a backup of one's key pair could be made by creating and encrypting an archive of this gpg filesystem tree, a far better solution is to use its integrated procedure, as the structure of its internal state may change from a version/platform of gpg to another. So the best course of action is to use the following command in order to generate a backup of a key pair in a standard, durable form:

```
$ gpg -o $(date '+%Y%m%d')-full-key-backup-for-james.bond-at-mi6.org.gpg --export-secr
```

[21] Other common extensions are `.gpg` (for encrypted content and also standard signatures), `.asc` (for clear-text signatures and other ASCII content), and `.sig` (for detached signatures).

This will produce a half-kilobyte file containing the full key pair, whose type is:

```
20211126-full-key-backup-for-james.bond-at-mi6.org.gpg: OpenPGP Secret Key Version 4,
```

Of course, so that it may be used in the future, this backup of (notably) the private key should *not* be encrypted with that same key.

Specifying in filenames the email address may be avoided, in the sense that rather than having multiple keys (ex: as many as email accounts), it is often more convenient to have a single key supporting multiple names/addresses (see the section about subkey below); so:

```
# If using fingerprints and potentially having multiple registered email
# accounts,just focusing on their common identity:
#
$ gpg -o $(date '+%Y%m%d')-full-key-backup-for-james.bond.gpg --export-secret-keys C3
```

A backup of the revocation certificate shall be done as well (knowing that by design it is not password-protected, and thus having access to this certificate is sufficient to be able to kill your key), preferably in a different location as the role of this certificate is to serve as an urgent safety measure should the private key be lost (non-emergency revocations should be performed thanks to the more adapted and informative `--generate-revocation` option instead).

For long-term auxiliary storage, such a backup can be printed (on paper), possibly thanks to Paperkey (installed on Arch with `pacman -Sy paperkey`). For example:

```
# To print directly:
gpg --export-secret-key my_key_fingerprint | paperkey | lpr

# To store first (less secure):
gpg --export-secret-key my_key_fingerprint | paperkey --output my_key_fingerprint.asc
```

Such exports are ASCII texts, but they can also take the perhaps more convenient (and maybe less secured if having to trust one's smartphone) form of a QR code:

```
$ gpg --export-secret-key my_key_fingerprint | paperkey --output-type raw | qrencode
```

Besides key pairs, following backups shall be done:

- the known public keys, thanks to: `gpg -o $(date '+%Y%m%d')-known-public-keys.gpg --export`
- the associated level of trust (level per public key): `gpg --export-ownertrust > $(date '+%Y%m%d')-openpgp-trust.txt`

How Can Public Keys be Shared? As mentioned, public keys can be freely shared without involving any specific risk, as in practice a private key cannot be derived from its public counterpart.

So basically any means of sharing them is legit, including the least secured ones. However the point is that their recipients must be sure that they obtained the right public certificate, and not one that has been tampered with.

Indeed, any man-in-the-middle M between peers A and B able to intercept the communication of A's public key could replace it by his. B would then have no means of detecting that it is actually relying on M's keys rather than on A's ones.

So, on top of the generation of key pairs, a safe mechanism to share public ones shall be carefully considered, to establish the authenticity of the binding between a public key and its owner. Such mechanisms exist in two forms, peer-to-peer ones, or centralised ones.

Decentralised Sharing The Web of trust is a decentralized trust model, which - like Internet federates a large number of computer networks - is to federate trust networks.

A user may have multiple key pairs, and each of the corresponding public keys may be known of various trust networks.

The trust conceded by identity A to identity B means that A endorses the association of the public key of B with the person or entity listed in its certificate.

The goal is to enable the emergence of some level of global trust from the trust that each given identity concedes to the various identities that it knows directly.

Trust is indeed to be spread, by extending it from peer to peer (or friend to friend) in an increasingly large network of trust, typically with trust levels that decrease with the number of peers that have to be traversed in the network before reaching a given identity: you may trust friends of your friends, albeit probably a bit less than your direct friends; networks of trust may reflect that increasing risk, typically based on mean shortest distance between endpoints.

In practice, if A expresses some level of trust to B, A will digitally sign (thus with its own private key) the public certificate of B, to assess its association with the identity it embeds. This is commonly done at key signing parties (a nice way of meeting likely-minded folks as well).

Various schemes for vetting (validating in practice the identity carried by B; ex: should we request B to show their identity card, to prove they control a given domain, or any other identity/ownership proof?) and voting (to decide on the overall trust to be derived from a potentially conflicting set of peer-to-peer endorsements A1, A2, etc. about B) exist; one remains of course free to decide for oneself on which grounds one concedes trust, it is the beauty of a decentralised mode of operation.

In practice, the sharing of public certificates used to be done through SKS key servers; it is as simple as requesting gpg to send the pblic key that corresponds to the specified fingerprint (here its last 8 characters):

```
$ gpg --send-keys E115A8B5
gpg: sending key 9D60ADA5E115A8B5 to hkps://keyserver.ubuntu.com
```

Note that this sharing discloses the corresponding email address, and thus exposes it to spam.

As various issues threaten SKS-based solutions, public keys may also be sent to the Hagrid-based OpenGPG server, `keys.openpgp.org` (which is not replicated to peer servers, yet performs more verification of the issuer of registered certificates).

To do so, register first this server in your configuration:

```
$ echo "keyserver hkps://keys.openpgp.org" >> ~/.gnupg/dirmngr.conf

# Reload gpg daemon:
$ gpgconf --reload dirmngr

# Extract the public key of interest in a .pub file:
$ gpg -o $(date '+%Y%m%d')-james.bond-at-mi6.org.pub --export james.bond@mi6.org
```

This file shall be uploaded via this web page that will guide you through the verification process, i.e. sending an email to the electronic address embedded in the transmitted public key in order to check that it is legit (by waiting for you to visit the URL that it generated and specified in said email).

More generally, various keyservers are looked up by gpg and thus can be considered (with different configurations regarding federation, verification, ability to forget keys, etc.).

Afterwards anyone will be able to search for such key:

```
$ gpg --search-keys james.bond@mi6.org
gpg: data source: https://keys.openpgp.org:443
(1)     James Bond <james.bond@mi6.org>
          512 bit ECDSA key 9A60ADA4E151B8B5, created: 2021-11-26
```

Of course checking that only one matches is returned is important to detect spoofing attempts.

Specifying your OpenPGP fingerprint in your email footers offers little interest, as your recipients cannot be sure that such incoming emails have not been tampered with.

So ultimately one will have either to trust such a decentralised scheme, or to trust a central authority like discussed next.

Centralised Sharing A centralized trust model is based on a Public Key Infrastructure (PKI, usually based on the X.509 standard), which relies exclusively on a Certificate Authority (CA), or more often a hierarchy of such: a CA's certificate may itself be signed by a different CA, all the way up to a self-signed root certificate.

So a certificate chain has to be validated, knowing that tools like browsers, and operating systems alike, come with their own keystore already comprising root certificates, and regularly updating them.

These certificates are well protected, yet any compromising thereof may jeopardise their whole "subtree".

Sharing Largely So a public certificate can be spread as widely as wanted, through key servers / PKIs, but also it should be shared through any reliable,

authoritative reference of a given identity, like one's own webserver, emails, social accounts, etc.

This can be directly your public certificate (here is mine)[22] or a (shorter) fingerprint thereof (ex: the full fingerprint of my key is `DCA8E181DC3CEAF0EAE4033F9987EE77188E9BF4`).

Such public keys can be listed and then obtained respectively thanks to:

```
$ gpg --list-keys james.bond@mi6.org
pub   brainpoolP512r1 2021-11-26 [SCA] [expires: 2022-11-26]
      C3987680AD9B79FDC6B7D25C9D60ADA5E115A8B5
uid           [ultimate] James Bond <james.bond@mi6.org>

# For a binary version of the public key:
$ gpg -o james-bond.pub ---export C3987680AD9B79FDC6B7D25C9D60ADA5E115A8B5

# For an ASCII-based version (ex: suitable to register in GitHub):
$ gpg -o james-bond.pub.asc --armor --export C3987680AD9B79FDC6B7D25C9D60ADA5E115A8B5
```

What can be done with these keys? One may:

- **encrypt** a file: `gpg -r james.bond@mi6.org -e my_file_to_encrypt`; this generates a `my_file_to_encrypt.gpg` file
- **sign** a file, with three possibilities:
  - `--sign` / `-s` to generate a file containing both the input file (wrapped in an OpenPGP packet) and the signature
  - `--clear-sign` to generate a file containing both the input file (verbatim, expected to be a text file) and the signature
  - `--detach-sign` / `-b` to only generate a file containing said signature; so the input file will be needed in this mode to verify that signature; this possibility is useful when distributing content (ex: binaries), so that the intended public can check the signature if wanted
- **decrypt** and possibly in the same movement **check the signature** of a file: `gpg -d my_file_to_decrypt.gpg` (everything will be output to the standard stream)
- **verify** a signature: see the `--verify` option for the 3 types of signatures
- **verify** signed emails:
  - import the public key of the sender: `gpg --search-keys dr.no@foobar.org`
  - determine whether it is valid and, more importantly, deserving trust (is it the right public key?); if yes; sign it with `gpg --edit-key dr.no@foobar.org`

[22] Note the HTTPS protection and that it currently refers to `online.fr` rather than to `esperide.com`.

- **import** keys (yours or not) in your email client; if using a (recent) Thunderbird, no plugin is needed, but the local gpg rings will *not* be used by Thunderbird; refer to this documentation
- **encrypt** and/or **sign** emails

A Link With Decentralized Identifiers

The use of key pairs in the absence of a certificate authority directly relates to Decentralized Identifiers (DIDs), a class of universal solutions (not depending on any context/organisation, and able to be recognized by any) with which anyone can create one's (globally unique) identifiers that remain in one's full control: one freely issues them, they remain valid as long as their issuer wishes (as none but their creator itself can revoke them), and (for example unlike mere UUIDs) they can be cryptographically verified by anyone.

No external central authority applies to such identifiers, which cannot reveal personal information unless decided by their issuer and thus sole controller.

In practice, although other solutions could maybe be considered, it involves, like discussed in the previous sections, generating on one's own at least a public/private key pair, to store safely the private one and to share as widely as needed the public one. Then one can sign and/or encrypt one's messages with a pretty good hope that they will remain secure for a while; such a system enables partial disclosure (as one chooses what one encrypts or signs) in full control (as all operations are driven by the private key that the issuer is the only one to control).

These decentralised identifiers, together with the principle of addressing a digital content by its fingerprint (ex: SHA1), offer a solution bringing many interesting properties and opening new possibilities to distributed systems (ex: for blockchains, a user account is often identified by the fingerprint of its associated public certificate).

Hints

- whenever useful, add the `--armor` option to use ASCII output armor, suitable for copying and pasting content in text format
- if you have multiple email accounts, thanks to `--edit-key` you can add each one of themin the same key as an identity (name), using the `adduid` command; you can then set your favourite one as primary
- to always show full fingerprints of keys, add `with-fingerprint` to your configuration file (typically `~/.gnupg/dirmngr.conf`)
- these Debian guidelines describe a robust, well-defined process for key management that may apply to most developers

See Also

- a complete, well-written tutorial, in French: Bien démarrer avec GnuPG
- other interesting usage hints, still in French

- GnuPG on Arch, for much additional information
- Network Management information

About Build Tools

Organisation: Copyright (C) 2021-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Friday, April 8, 2022

Purpose of Build Tools

A build tool allows to automate all kinds of tasks, by **applying rules and tracking dependencies**: not only compiling, linking, etc. applications, but also checking them, generating their documentation, running and debugging them, etc.

Choice

Often build tools are tied to some programming languages (ex: Maven for Java, Rebar3 for Erlang, etc.).

Some tools are more generic by nature, like late GNU autotools, or Cmake, GNU make, etc.

For most uses, our personal preference goes to the latter. Notably all our Erlang-based developments, starting from Ceylan-Myriad, are based on GNU make.

GNU make

We recommend the reading of this essential source for reference purpose, notably the section about The Two Flavors of Variables.

Taking our Erlang developments as an example, their base, first layer, Ceylan-Myriad, relies on build facilities that are designed to be also reused and further adapted / specialised / parametrised in turn by all layers above in the stack (ex: Ceylan-WOOPER).

For that, Myriad defines three top-level makefiles:

- base build-related *variables* (settings) in GNUmakevars.inc, providing defaults that can be overridden by upper layers
- *automatic rules*, in GNUmakerules-automatic.inc, able to operate generically on patterns, typically based on file extensions
- *explicit rules*, in GNUmakerules-explicit.inc, for all specific named make targets (ex: `all`, `clean`)

Each layer references its specialisation of these three elements (and the ones of all layers below) in its own GNUmakesettings.inc file, which is the only element that each per-directory `GNUmakefile` file will have to include.

Such a system allows defining (build-time and runtime) settings and rules once for all, while remaining flexible and enabling individual makefiles to be minimalistic: beside said include, they just have to list which of their subdirectories the build should traverse (thanks to the `MODULES_DIRS` variable, see example).

See Also

asdf, an extendable version manager for various languages (Ruby, Node.js, Elixir, Erlang, etc.).

One may refer to the development section of Ceylan-Hull, or go back to the Ceylan-HOWTOs main page.

Version Control Systems: in Practice, now, Git

Organisation: Copyright (C) 2021-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Tuesday, June 28, 2022

Overview

No real software development shall happen without the use of a VCS - standing for *Version Control System* - of some sorts, notably in order to track the versions of the source files involved and to ease the collaborative work on them.

Many solutions have been defined for this purpose (CVS, Clearcase, SVN, Mercurial, etc.), but now a single tool is the de facto standard: Git, which is a distributed version control system available as free software; refer to its website for more details.

Git Usage

Beyond the documentation relative to its general use, projects have to adopt their own set of conventions - regarding the management of branches, commits, tags, etc - based on their preferences and context.

Recommended General Conventions

The ones to which we try to stick are:

- the path (including names) of files and directories should not include spaces; as separator, prefer dashes (`-`) to underscores (`_`)
- the character case shall be uniform (ex: directory names starting always or never with a capital letter)
- the language used shall be uniform (ex: only English)
- a commit message shall describe synthetically the modifications operated on the corresponding new filesystem snapshot; such a message, preferably in English, should always start with a capital letter and end with a dot; ex: "*Fixed the computation of angles.*"
- the file formats shall be, as much as possible, homogeneous, notably for text files with regard to the line terminators; either only UNIX conventions (only `LF`; preferred), or only Windows ones (`CRLF`); use `dos2unix` whenever necessary, possibly automated through a Git hook
- abbreviations are convenient (ex: `br` for branch, `co` for `checkout`, etc.); they can be defined in one's `~/.gitconfig`

Basic Operations

- **managing branches**:
  - **creating** branches is done thanks to the `checkout` command, often abbreviated as `co` here
  - to create a branch deriving from the current one (the current `HEAD`) and switching to it at the same time (performing its co, while inheriting any local changes): `git co -b my_new_branch`
  - to create a local branch corresponding to a remote one (let's suppose it is named `some_branch`), assuming that a remote server (ex: `my_remote`, possibly `origin`) has already been declared (ex: `git remote add my_remote URL`):
    - first step is to update the remote-tracking branches with `git fetch my_remote`, then to create the target local branch tracking that remote (upstream) one with: `git co -b some_branch my_remote/some_branch` (also switching to it here)
    - a shortcut is to use `git co --track my_remote/some_branch` instead
    - even shorter, if the name of the target local branch name does not exist yet and matches exactly matches a name on only one remote, `git co some_branch` will suffice
  - to delete a local branch: `git branch -d my_branch`
- **managing tags**:
  - to list (local) tags: `git tag`
  - to have information about an already-existing tag: `git show my_tag`
  - to set a new annotated tag: `git tag -a foobar-version-2.4.0 -m "Release of the version 2.4.0 of Foobar."`; prefer naming tags differently from branches (ex: `foobar-version-2.4.0` rather than `foobar-2.4.0`) to spare ambiguities to Git
  - a set tag must be specifically pushed on a remote, for example: `git push origin my_tag`; all tags can be pushed with `git push --tags` (the remote can be implied)
  - to delete a tag that was not pushed: `git tag --delete my_tag`
- **determining whether a file is in VCS**, knowing that due to `.gitignore` rules, `update-index --skip-worktree`, etc. it is not always obvious:

```
# Target file is tracked iff is listed by:
$ git ls-files | grep my_file

# Or, in order to trigger an error if this target file is not tracked:
git ls-files --error-unmatch my_file
```

- getting the **version of a file as it was at a given commit**:

```
# Replaces the current version of that file by the designated one:
$ git checkout COMMIT_ID path/to/the/target/file

# Outputs on the console the designated version:
$ git show COMMIT_ID:path/to/the/target/file

# Outputs on the console the diff between the designated
# version and the current one:
$ git show COMMIT_ID path/to/the/target/file
```

Managing Branches

Creating branches allows to separate threads of work (while preserving their lineage) and progress concurrently. Yet often their content will have to converge ultimately; depending on the intent, two use cases can be considered, resulting in different Git uses.

Merge versus Rebase Here one may want:

- either to **integrate back a development branch** (ex: `my-feature`) **in a shared, parent one** (ex: `master`): then one shall prefer using `merge`, in order to keep separate histories and not affect the past one of the shared branch
- or to **resynchronise a development branch** (ex: `my-feature`) **on the last version of a shared branch and continue these developments**: then one shall prefer `rebase`, so that the history of the development branch contains only its own changes (less noise, linear history)

In practice, in order to transfer the changes of a branch A in a branch B:

```
$ git co B

# Either first case (integrate development A in master B):
$ git merge A # or: git pull A

# Or second one (resynchronise development B on master A):
$ git rebase A # or: git pull -rebase A
```

How such a last `rebase` of branch A in branch B is done? The bifurcation point of B compared to A is moved from its initial position to the current head of A, on which all changes recorded in B are applied; the resulting history of B looks like if these changes had been directly performed from the version of A designated in this rebase, and thus B can be then directly fast-forwarded to its tip, which comprises both the changes synchronised from A and, then, the ones specifically introduced in B.

Then, to update the remote with these post-rebase commits, `git push --force-with-lease` shall be used[23].

[23] Rather than just performing just a push, having it fail, pulling, and ending up with duplicates of the changes. Should this happen, rewind these changes, for example with: `git reset --hard <full_hash_of_commit_to_reset_to>`.

More information: [1] or, in French: [2], [3], [4].

Directly Transferring Changes Sometimes, one may want to directly transfer the changes of a derivate branch B in a parent branch A. When one knows for sure that the versions in B shall be preferred in all cases to their counterparts in A (note that a classical merge is already fully able to manage fast-forwards), one may use:

```
$ git checkout A
$ git merge -X theirs B
```

No conflict should arise (source).
The same is possible with `rebase`; for example: `git rebase -X theirs B`.
Note that `-X` a *strategy* option, whereas `-s` would be a *merge strategy* option.
Using here `ours` rather than `theirs` :

- `-X ours` uses "our" version of a change *only when there is a conflict*
- whereas `-s ours` ignores the content of the other branch entirely (in all cases), and use "our" version instead

Another way of forcing the content of a branch B to be the same as the one of a branch A is, while B is checked-out, to execute: `git reset --hard A`. As mentioned previously, push shall be done then with `git push --force-with-lease`.

Common Procedures

Overcoming auto-signed SSL certificate issues To avoid, typically in a company internal setting, errors like:

```
Cloning into 'XXX'...
fatal: unable to access 'https://foo.bar.org/XX/XXX/': SSL certificate problem: self s
```

the `http.sslVerify=false` option may be used, even if it weakens the overall security.
This is typically useful initially:

```
$ git -c http.sslVerify=false clone https://foo.bar.org/XX/XXX
```

In order that the next operations (ex: future pushes) overcome too this problem for the current repository, use from within the current clone:

```
$ git config http.sslVerify false
```

Setting the right metadata for the next commits Doing so prevent from having to amend commits a posteriori.
If these information apply for all projects:

```
$ git config --global user.name "John Doe"
$ git config --global user.email john.doe@foobar.org
```

Otherwise shall be done at least on a per-project basis with:

```
$ git config user.name "John Doe"
$ git config user.email john.doe@foobar.org
```

Also `git config --global --edit` may be of use (beware to trigger a `vi` by accident...).

Performing operation on remotes with no systematic authentication Using a SSH key pair, hence with its public key declared on said remote, is a relevant approach, safer than from example using a `~/.netrc` file.

Creating an empty branch Rather than creating it from a pre-existing branch and removing all inherited content, prefer:

```
$ git checkout --orphan my_new_branch
```

(typically useful for GitHub Pages branches)

Listing differences with prior versions of a file In order to list the differences of a given file with the previous commits (precisely: of a set of pathspecs), one may use our `dif-prev.sh` script, which by default reports the differences with the last committed version. With the `--all` option, it lists all differences, until the first addition of this file.

Preventing the commit of a file in VCS that is often locally modified One should use this method:

```
$ git update-index --skip-worktree <file-list>
```

The opposite operation is:

```
$ git update-index --no-skip-worktree <file-list>
```

Listing the files managed in VCS from the current directory Use `git ls-files` to determine the files that are already managed in VCS, recursively from the current directory.

To list the untracked files (i.e. the files *not* in VCS), use `git ls-files --others`.

Reducing the size of a repository One may use our list-largest-vcs-blobs.sh script to detect any larger files that should not be in VCS (ex: should a colleague have committed by mistake a third-party archive, or unexpected data such as CSV files).

Then install BFG Repo-Cleaner:

```
$ mkdir -p ~/Software/bfg-repo-cleaner/
$ cd $_
$ mv ~/bfg-1.14.0.jar .
$ ln -s bfg-1.14.0.jar bfg.jar
# For example in ~/.bashrc:
$ alias bfg="java -jar ~/Software/bfg-repo-cleaner/bfg.jar"
```

All developers should be asked to commit their sources (git add + push), to archive their clone (ex: in a timestamped `.xz` file like `20220412-archive-clone-foobar.tar.xz`), and to wait until notified that they can create a new clone.

The repository may be then cleaned up (ex: from large, unnecessary CSV files) in isolation, with:

```
$ git clone --mirror XXX/foobar.git
$ bfg --delete-files '*.csv' foobar.git
$ cd foobar
$ git reflog expire --expire=now --all && git gc --prune=now --aggressive
$ git push
```

Then all developers shall be requested to perform a new clone and to check the fetched content (ex: with regard to the content of the last branch in which they committed).

Fixing LF vs CRLF End of Line Problems Use Git Attributes to specify proper files and paths attributes.

One may define a `.gitattributes` file for example with `*.js eol=lf`, `* text=auto`, or:

```
# No CRLF conversion for DOS/Windows batch files.
# They should be stored with the CRLF line terminators.
#
*.bat -crlf
```

Fixing a commit message If no push was done, it is as simple as replacing the former message by a new one, like in:

```
$ git commit --amend -m "This is a fixed commit message."
```

Tools

On Most Platforms

At least on UNIX, the command-line Git client (`git`) is certainly the best tool. In difficult situations, graphical tools such as `gitk` may be of help.

See also our Ceylan-Hull section about VCS-related scripts.

On Windows

Tools like TortoiseGit may foster a view on the usage of Git that is a bit particular, conflating concepts or introducing extra ones (ex: a `sync` command). Apparently also at least some pulls did not reintroduce files just removed from the working directory.

More generally, cloning on a Windows host an UNIX-originating repository comprising symbolic links may induce oddities (ex: a symlink named `S` pointing to `Foobar` resulting, on a Windows clone, in a file named `S` whose content is, literally, the text `"Foobar"`, instead of the expected content of the `Foobar` file).

Another option is to use Visual Studio Code (`vscode`), which supports natively Git (provided that the command-line version is already installed). One may select `View -> SCM` (or `Ctrl-Shift-G`) for that. Clicking on the "VCS" icon (three rings links by two curves; the third from the top) displays a contextual view offering various associated operations (here based on Git).

We finally preferred using MSYS2 + Git rather than Git Bash, named "Git for Windows"; hints to speed up these tools may apply.

Inner Workings

Git stores internally every version of every file separately (not as a diff with a parent version) as a blob (an opaque binary content) identified by its (SHA1) hash.

A commit is the identifier of a tree representing the filesystem of interest at a given moment (snapshot). This tree references the files through their SHA1, similarly to a Merkle tree.

A branch is thus nothing but a pointer on a given commit, and `HEAD` designates the current branch. Git stores natively only blobs, trees and commits.

The reported differences in the content of a file or a tree are thus only recreated (established dynamically) by Git commands, they are not natively tracked.

Translations

From English to French:

- `repository` -> `dépôt`
- `to checkout` -> `extraire`
- `to commit` -> `valider`
- `a commit` -> `validation`
- `in VCS` -> `en GCL` (Gestion de Configuration Logicielle)
- `snapshot` -> `instantané` (de l'état du sous-système de fichiers géré en GCL)
- `merge` -> `fusion`
- `head` -> `tête`
- `fast-forward` -> `avancement direct`
- `fast-forwarded` -> `directement avancée`

Documentation

Many pointers exist, doing a great job in unveiling how Git is to be used.

In English, Pro GIT is surely a reference.

In French:

- introduction en français

- cours sur OpenClassrooms
- référence incontournable et conseillée : Pro GIT, notamment pour l'explication de ses rudiments puis de son fonctionnement interne, à commencer par ses objets

Documentation Generation

Organisation: Copyright (C) 2022-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Wednesday, January 12, 2022

Lastly updated: Tuesday, August 30, 2022

Objective

We want to be able to generate, **from a single source**, at least two **documentation formats**:

- a set of **interlinked static web pages** (the most popular, flexible format)
- a single, **standalone PDF file** (convenient for offline reading, printing, etc.)

The document source shall be expressed in a simple, non-limiting, high-level syntax; in practice a rather standard, lightweight markup language.

All standard documentation elements shall be available (ex: title, tables, images, links, references, tables of content, etc.) and be customisable.

The resulting documents shall be quickly and easily generated, with proper error report, and be beautiful and user-friendly (ex: with well-configured LaTeX, with appropriate CSS, icons and features like banners, with proper rendering of equations).

Per-format overriding shall be possible (ex: to define different image sizes depending on web output or PDF one).

The whole documentation process shall be powered only by free software solutions, easily automated (ex: with Make) and suitable for version control (ex: with Git).

Our Recommended Approach

Principle

We chose to rely on the reStructuredText syntax and tools, also known as RST, a part of the Docutils project. We did not specifically rely on elements related to Python or the Sphinx toolchain.

We augmented reStructuredText with:

- a set of **make-based defines and rules** (automatic or explicit) that were aggregated in Ceylan-Myriad (see notably GNUmakerules-docutils.inc and the generate-docutils.sh generation script); this mechanism is layer-friendly, in the sense that all layers defined (directly or not) on top of Myriad are able by default to re-use these elements and to customise them if needed
- a **template** on which we rely for most documents, featuring notably a standard table (to specify usual metadata such as organisation, contact information, abstract, versions, etc.), a table of contents, conventions in terms of title hierarchy and, for the HTML output, a banner (a fixed, non-scrolling panel offering shortcuts, in the top-right corner of the page)

- a simple **tag-based system** to have the actual document markup (`*.rst`) directly generated from a higher-level source one (`*.rst.template`); in practice, if defined, only the latter element is edited by the user, and tags (such as `*_VERSION_TAG`, `*_DATE_TAG`, etc.) are automatically filled-in appropriately

Of course this website, and many others that we created, rely on this approach; as an example, one may look at the sources of the current document.

Specific Topics

Rendering Mathematical Elements With the RST toolchain, the PDF output, thanks to LaTeX, offers built-in high-quality rendering of mathematical elements such as equations, matrices, etc.

By default, the HTML output does not benefit from LaTeX, and remains significantly less pleasing to the eye, and less readable.

So we complement it by MathJax, a neat open-source "*JavaScript display engine for mathematics that works in all browsers*".

It shall thus be installed once for all first. For example, on Arch Linux, as `root`, it is sufficient to execute:

```
$ pacman -Sy mathjax
```

Then, to enable the use of MathJax for a given website based on Ceylan-Myriad, run from its root (often a `doc` directory):

```
$ make create-mathjax-symlink
```

(this target is defined in GNUmakerules-docutils.inc; it boils down to symlinking `/usr/share/mathjax`; see also the HOWTOs corresponding makefile to properly manage this dependency afterwards, notably when deploying web content)

The list of TeX/LaTeX commands supported by MathJax may be of use.

Each LaTeX command may either be specified directly inline, in the text (with `:math:‘LATEX_CMD‘`) or in a block indented after a `.. math::` directive.

This allows to define inline mathematical elements, like $P = \begin{pmatrix} 10 \\ 45 \end{pmatrix}$ (obtained with `P = \\begin{pmatrix} 10 \\\\ 45\\end{pmatrix}`) or standalone ones, like:

$$M = \begin{bmatrix} a11 & a12 & ... & a1n \\ a21 & a22 & ... & a2n \\ ... & ... & ... & ... \\ am1 & am2 & ... & amn \end{bmatrix}$$

obtained thanks to:

```
M = \begin{bmatrix}
        a11 & a12 & ... & a1n \\
        a21 & a22 & ... & a2n \\
        ... & ... & ... & ... \\
        am1 & am2 & ... & amn \\
    \end{bmatrix}
```

For $\phi : R \rightarrow]0,1[$ (i.e. `\phi: \mathbb{R} \rightarrow ]0,1[`), we may have $P_e = \phi(m + \phi^{-1}(P_n))$ is `P_e = \phi(m+\phi^{-1}(P_n))`.

If $\phi(x) = e^x/(1+e^x)$ (translating to `\phi(x) = e^{x}/(1+e^{x})`), then:

$$P_e = \frac{P_n.e^m}{1 + P_n.(e^m - 1)}$$

(translating to `P_e = \frac{P_n.e^{m}}{1 + P_n.(e^{m}-1)}`)

A few other examples of resulting math-related outputs can be seen in this section.

Title Hierarchy It must be consistent: a given type of subtitle must always be placed at the same level in the title hierarchy.

We rely on the markup conventions exposed in this demonstration file (created by David Goodger), whose source is here.

From the top-level title to the most nested ones:

- `=`, on top and below the title (document title)
- `-`, on top and below the title (document subtitle)
- `=`, below the title (H1)
- `-`, below the title (H2)
- `.` , below the title (H3)
- `_`, below the title (H4)
- `*`, below the title (H5)
- `:`, below the title (H6)
- `+`, below the title (H7)

Image Sizes Responsive images, i.e. images that automatically adjust to fit the size of the screen, can be used. They are then defined for example thanks to:

```
<img src="foobar.png" id="responsive-image-large"></img>
```

Various standard sizes have been defined, all prefixed with `responsive-image-`; from the biggest (95%) to the smallest (10%), as defined for example in myriad.css, they are: `full`, `large`, `intermediate`, `medium`, `reduced`, `small`, `tiny`, `xsmall`.

Multi-File Documents

Targeting a Standalone Document Although they tend to be less convenient to edit, longer documents may be split in a **set of RST source files** (the Myriad documentation is an example of it; the WOOPER documentation is an example of the opposite approach, based on a single source file).

Targeting Interlinked Modular Documents In some cases, at least for the HTML output, the need is not to produce a single, large, monolithic document, but a set of interlinked ones (the present HOWTO is an example thereof) that can be browsed as separate pages.

Then a convenient approach is to define different entry points for different output formats, like, for these HOWTOs, this one for the HTML output and this one for the PDF output.

Inner Links Defining any title (ex: the `"Rendering Mathematical Elements"` one above) automatically introduces in turn a corresponding anchor, which, for the HTML output, can then be referenced from any page, for example as raw HTML (like MyPage.html#rendering-mathematical-elements, or directly from the current page as #rendering-mathematical-elements) or directly through RST in the document (ex: specified as `‘Rendering Mathematical Elements‘_`, resulting in: Rendering Mathematical Elements).

Note the light transformation (spaces becoming dashes) of the specified name once a it is translated into a legit HTML anchor.

Extra local anchors (ex: that could be named `"how to render equations"`) can also be specified anywhere in the document (ex: just before the previously mentioned title, so that it can be designated with other words), thanks to:

```
.. _‘how to render equations‘:
```

It can then be referenced from the same page as #how-to-render-equations or from another one as MyPage.html#how-to-render-equations.

Note that titles and hypertext links introduce local links as well, so one's inner links may clash with them (resulting in `(ERROR/3) Duplicate target name [...]`); the best option is generally to phrase these inner links differently.

Commenting To comment-out a block of text, just add `..` at the beginning of a line, then, from the next line, put that block, indented of at least one space; this must be a legit block (see Defining Blocks).

Defining Blocks All lines of a block shall start with the same whitespace. So, whenever a given block is not left-justified (at least one of its lines starts with a different offset), prefer having all lines of such a block be indented of (at least) 4 spaces (i.e. a tabulation).

Otherwise, if using a single space to indent, as soon as a line of the block is to start with 3 spaces, whitespace-cleanup operations will combine them with the first one to form a tabulation (4 spaces in a row), and all lines of the commented block will not start with the same whitespaces, which could result, from the point of view of RST tools, in an invalid block.

Before a code block (e.g. introduced with `.. code:: erlang`), a single colon should be used, not two of them. For example:

```
This algorithm can be:

.. code:: erlang
   [...]
```

Miscellaneous

These hints apply more generically than only with a RST toolchain.

Validating / Checking HTML Content

In addition to the verification of the messages reported when the document is built, some tools allow to perform some checks on a generated document.

Notably an online HTML page, or set of pages, can be verified by third-party tools like this one, to detect dead links.

Pointing to a Specific Moment in a Linked Video

It is as simple as designating, in an HTML link, the targeted second by suffixing the URL video filename with `#t=DURATION_IN_SECONDS`, like in `some-video.mp4#t=1473`[24].

Conversion between Markup Formats

Pandoc is the tool of choice for such operations, as it often yields good results.

For example, in order to convert a page written in Mediawiki syntax, whose source content has been pasted in a `old-content-in-mediawiki.txt` file, into one that be specified in a GitLab wiki (hence in GFM markup, for *GitLab Flavored Markdown*) from a converted content, to be written in a `converted-content.gfm` file, one may use:

```
$ pandoc old-content-in-mediawiki.txt --from=mediawiki --to=gfm --standalone -o conver

# Or, for older versions of pandoc not supporting a gfm writer:
$ pandoc old-content-in-mediawiki.txt --from=mediawiki --to=markdown_github --standalo
```

Then the content in `converted-content.gfm` file can be pasted in the target GitLab wiki page.

Another example is the conversion of a GitLab wiki page into a RST document (ex: then for a PDF generation):

```
$ pandoc my-gitlab-wiki-extract.gfm --from=gfm --to=rst --standalone -o my-converted-d

# Or, for older versions of pandoc not supporting a gfm reader:
$ pandoc my-gitlab-wiki-extract.gfm --from=markdown_github --to=rst --standalone -o my
```

Finally, if really needing to generate a Word document, an example may be:

```
$ pandoc my-document.rst --from=rst --to=docx -o my-converted-document.docx
```

The lists of the input and output formats supported by Pandoc and of their corresponding command-line options is specified here.

[24] With `mplayer`, use the `o` hotkey to display elapsed durations.

These options are also returned by: `pandoc --list-input-formats` and `pandoc --list-output-formats` (or, for older versions of pandoc, thanks to `pandoc --help`).

An input file may not be encoded in UTF-8, which can result in:

```
pandoc: Cannot decode byte '\xe9': Data.Text.Internal.Encoding.Fusion.streamUtf8: Inva
```

In this case, the actual encoding shall be determined, for example with:

```
$ file input.html
input.html: HTML document, ISO-8859 text
```

Then the encoding may be changed before calling pandoc, for example like:

```
$ iconv -f ISO-8859-1 -t utf-8 input.html | pandoc --from=html --to=markdown_github --
```

Transformation of PDF files

For that, one may use the `pdftk` tool, possibly with the `convert` one, which comes from ImageMagick (typically available thanks to a `imagemagick` package):

- to **split all pages of a PDF** in as many individual files (named `pg_0001.pdf`, `pg_0002.pdf`, etc.): `pdftk document.pdf burst`
- to **convert a PDF file** (typically a single page) into a **PNG** one (typically in order to edit the PNG with The Gimp afterwards): `convert pg_000x.pdf pg_000x.png`
- to convert (possibly back) a **PNG file to a PDF** one: `convert pg_000x-modified.png pg_000x-modified.pdf`
- to **concatenate PDFs**: `pdftk 1.pdf 2.pdf 3.pdf cat output 123.pdf`

Image Transformations

One may rely on:

- GIMP (*GNU Image Manipulation Program*; corresponding, on Arch, to the `gimp` package)
- or on command-line ImageMagick (on Arch, the `imagemagick` package, which provides notably the `convert` and `display` executables)

To invert/negate an image (swap colors with their complementary ones, while preserving alpha coordinates):

```
$ convert source.png -channel RGB -negate target.png
```

See also the Myriad's automatic rules, which generate `X-negated.png` from `X.png` thanks to: `make X-negated.png`.

UML Diagrams

If SysML can also be of interest, we focus here on UML2 class diagrams (one of the 14 types of diagrams provided by UML2).

Quick UML Cheat Sheet

Multiplicities A **multiplicity** is a definition of *cardinality* (i.e. number of elements) of some collection of elements.

It can be set for attributes, operations, and associations in a class diagram, and for associations in a use case diagram. The multiplicity is an indication of how many objects may participate in the given relationship.

It is defined as an inclusive interval based on non-negative integers, with `*` denoting an unlimited upper bound (not, for example, `n`).

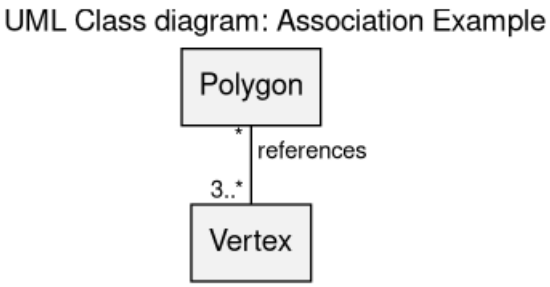
Most common multiplies are:

- no instance or one instance: `0..1`
- any number of instances, including zero: `*` (shorthand for `0..*`)
- exactly k instances: `k` (so, if k=5, `5`)
- at least M instances: `M..*` (`2..*`)
- at least M instances, but no more than N (hence bounds included): `M..N` (e.g. `3..5`)

For associations, the default multiplicity is automatically is `0..1`, while new attributes and operations have a default multiplicity of `1`.

Association An **association** is a relation between two classes (*binary association*) or more (*N-ary association*) that describes structural relationships between their instances.

For example a polygon may be defined from at least 3 vertices that it would reference, whereas a point may take part to any number of polygones (including none):



(see the sources of this diagram)

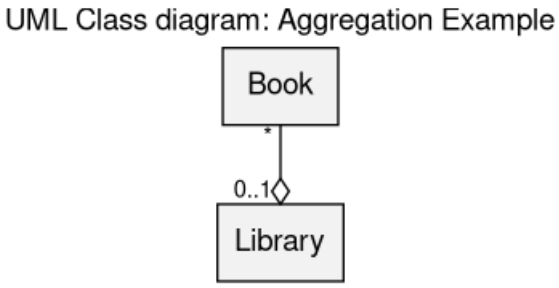
The **multiplicity** of an endpoint denotes the number of instances of the corresponding class that may take part to this association. For example, at least 3 points are needed to form a polygon, whereas any number of polygons can include a given point.

In UML the **direction** of the association is easily ambiguous (here we have to rely on external knwoledge to determine whether a polygon is composed of points, of if a point is composed of polygons). Adding a chevron (like `>` or `<`, e.g. `"references >"` ; ideally this should be a small solid triangle) to the text

is not a good solution either, as the layout may place the respective endpoints in any relative position. Adding an arrow to the end of the line segment cannot be done either, as it would denote the *navigability* of the association instead.

Aggregation An **aggregation** is a specific association that denotes that an instance of a class (ex: `Library`) is to loosely contain instances of another class (ex: `Book`), in the sense that the lifecycle of the contained classes is not strongly dependent on the one of the container (ex: books will still exist even if the library is dismantled).

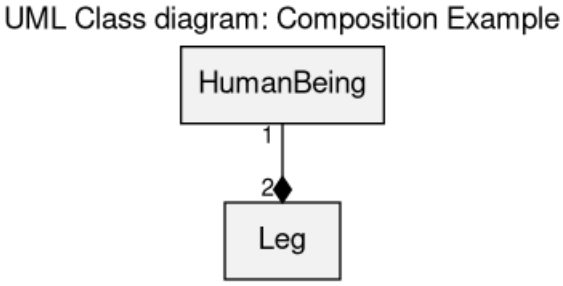
Here a library may contain any number of books (possibly none), and a given book belongs to at most one library.



(see the sources of this diagram)

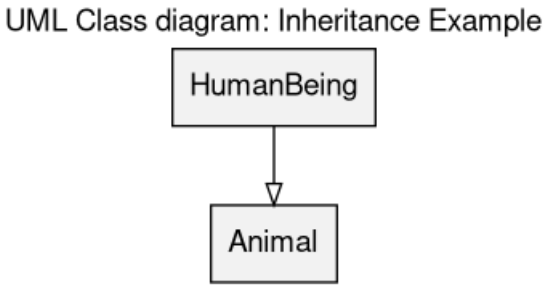
Composition A **composition** is a specific association that denotes that an instance of a class (ex: `HumanBeing`) is to own instances of another class (ex: `Leg`), in the sense that the lifecycle of the contained classes fully depends on the one of the container (here, if a human being dies, his/her legs will not exist anymore either).

Here a human being has exactly 2 legs, and any given leg belongs to exactly one human being (therefore this model does not account for one-legged persons).



(see the sources of this diagram)

Inheritance An **inheritance** relationship is a specific association that denotes that a class (ex: `HumanBeing`) is a specific case of a more general one (ex: `Animal`), and thus that an instance of the first one is also an instance of the second one ("is-a" relationship).



Here a human being is a specific animal.
(see the sources of this diagram)

Tooling In a design phase, one may prefer lightweight tools like Graphviz, PlantUML or even Dia.

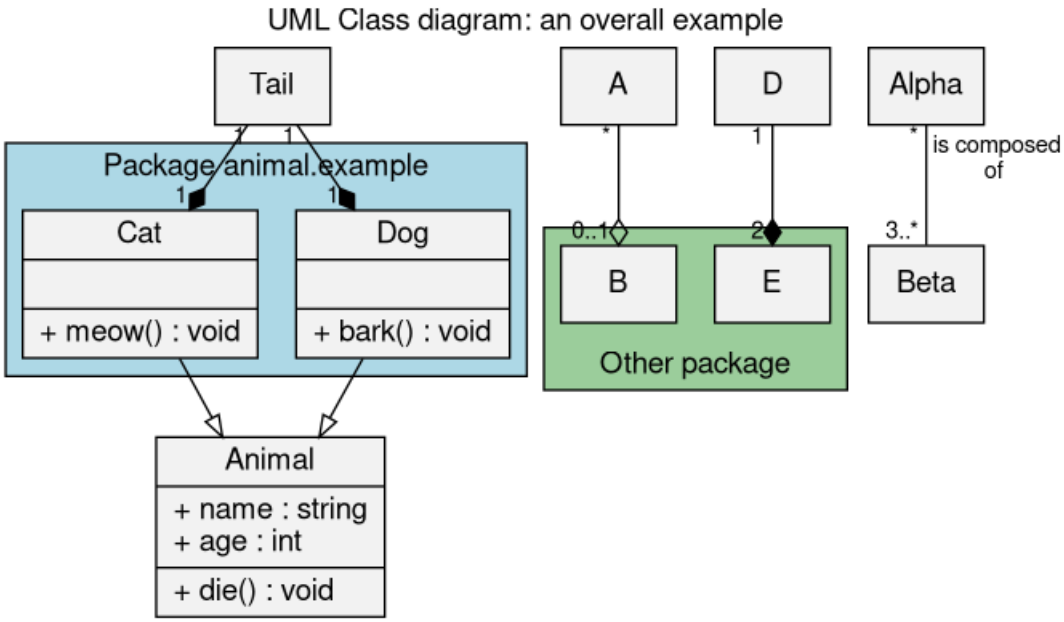
As long as the architecture of a framework is not stabilised, having one's tool determine by itself the layout of the rendering (rather than having to place manually one's graphical components) is surely preferable.

For that we use Graphviz, with our own build conventions.

For example, supposing this diagram example, i.e. a source file named `uml_class_diagram_example.graph`:

```
$ make uml_class_diagram_example.png
# or, to force a regeneration and a displaying of the result:
$ make clean uml_class_diagram_example.png VIEW_GRAPH=true
```

This example results in the following diagram:



(see the sources of this diagram)

Using Additional Fonts

One may use websites like Dafont in order to select, based on appearance and licence, a given TTF font.

At least on Arch, it is sufficient to copy the corresponding downloaded TTF file (as root) in `/usr/share/fonts/` so that tools like The Gimp support it right afterwards (ex: no need to run `fc-cache` beforehand).

Data Management

Organisation: Copyright (C) 2022-2022 Olivier Boudeville

Contact: about (dash) howtos (at) esperide (dot) com

Creation date: Saturday, November 20, 2021

Lastly updated: Tuesday, August 30, 2022

Overview

This section concentrates information about **data management**, including data **formats** and data **processing tools**.

General-Purpose Data Format

Such a format is typically useful to hold configuration information.

We prefer JSON to, for example, YAML, due to the Python-style indentation on which the latter relies in order to indicate nesting.

Language-Independent Data Formats

JSON A JSON document is in plain-text and may contain:

- basic types:
  - Number: `2` or `4.1`
  - String: `"I am a string"`
  - Boolean: `true` or `false`
  - `null`: to denote an empty value
- attribute-value pairs (ex: `"firstName": "John"`)
- "arrays" (ordered lists), ex: `"myNumbers": ["12", "7", "4"]`
- "objects" (collection of name-value pairs), ex:

```
{

  "address": {
     "streetAddress": "21 2nd Street",
     "city": "New York"
  }

}
```

The order in arrays is expected to be preserved, but not the one of the elements in an object.

Defining an element (ex: an attribute-value pair) more than once is allowed, and the last instance thereof will be the one kept.

Ex:

```
{
 "tcp_port": 8084,
 "tcp_port": 8085,
 [...]
}
```

Here, once the document is parsed, `tcp_port` will be considered equal to `8085`.

Pretty-Printing On GNU/Linux, one may rely on `jq`, a command-line JSON processor.
Ex: `jq . my_document.json`.

Validating One may consider that a given document is a legit JSON one iff `jq type` reports a non-empty output.
Example:

```
$ jq type my_document.json
"object"
```

Example Regarding syntax, a typical JSON document is:

```
{
  "firstName": "John",
  "__comment": "This is a comment!",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Specifying comments With JSON, there is, on purpose, no built-in way to add comments.

The sole solution/workaround is to add comments as specific fields, although they will end up as data like the other fields.

We recommend to mark them specifically (ex: as `__comment`) so that they should not interfere with the "real" data. As an example, see the second key of the previous JSON document.

YAML YAML is a data serialization language for all programming languages.

We prefer the `.yaml` extension to the `.yml` one.

No tabulation should be used for indentation, only spaces, and preferably a fixed amount of them; we used to prefer 4, now 2, since it allows to properly align the items listed with a dash (ex: `"- I am an item"`).

With Emacs, the Yaml Mode may be of help.

Erlang-Friendly Data Format: ETF

Such a format is typically useful to hold configuration information in an Erlang context.

We recommend the use of ETF (the *Erlang Term Format*), that we find particularly useful and even more suitable than JSON (entry order preserved, comments supported, etc.).

Data-related Processing Tools

Whenever needing to perform **numerical operations** on data, we recommend the use of Scilab or GNU Octave, which are the two major open-source alternatives to MATLAB.

As such, the three of them support mostly the same syntax (even if Scilab puts less emphasis on syntactic compatibility with MATLAB than Octave does).

From now on, the specific tool being used among the two MATLAB alternatives will not be specifically mentioned (mentioning "the tool" instead). We tried both alternatives and had more issues (build/installation, proper display) with Scilab, so we mostly used Octave.

Common Hints About Scilab and Octave

Syntax Putting a semicolon at the end of statement prevents the console from printing the corresponding value (answer, variable assignment, etc.).

Array-based Functions Many functions are defined so that they can be called also with *arrays* of parameters, instead of just standalone values. For that, a dot/period-based syntax has been introduced so that each element of an input vector is applied to the function in turn. A key understanding is that such a dot applies not to variables but to operators; for example `2*x.^2+1` shall not be read as `2*(x.)^2+1` but as `2*x(.^)2+1`.

As an example, a ϕ function can be defined[25] as $\phi(x) = e^x/(1+e^x)$:

```
function retval = phi (x)
 retval = exp(x)./(1+exp(x));
endfunction
```

Then this function can be called either with a standalone value:

```
phi(1)
ans = 0.7311
```

or with an array thereof (a vector):

```
phi([1,2])
ans =
  0.7311   0.8808
```

Outputs Beware to the lower-precision of textual outputs, which may be misleading (ex: use `format long` with Octave to request 15 significant figures).

Script Files Rather than being directly interpreted (ex: from a pasted text), a series of statements can be gathered in a script file (a bit different from a function file[26]) that can be loaded and executed afterwards.

Their conventional extension in `m` (ex: `foobar.m`), due to the MATLAB legacy.

Avoid dashes in the script filenames, as it may be interpreted as `minus`; prefer underscores (ex: `phi_exp.m` rather than `phi-exp.m`).

An example of script, named `phi_exp.m`:

```
# An initial comment prevents Octave from thinking that this
# is a function file:
1;

# The function name can be freely chosen, it does not have to
# correspond to the script filename:
#
function retval = phi_exp (x)
   retval = exp(x) ./ (1+exp(x))
endfunction

xs = -10:10
ys = phi_exp(xs)

plot(xs,ys)

title ("A function phi to map values V to a probability-suitable ]0,1[ interval")
ylabel ("\phi(V")
xlabel ("V")
grid on
```

[25] Note that, for basic operation like additions, the `.+` operator has been deprecated in favor of just `+`.

[26] A so-called "function file" is one that starts with a function definition and that must only be called from a "script file".

Provided that such a script is located in a well-known directory of the tool (typically its working directory), it can be executed simply by entering its filename without extension; for example:

```
octave:1> phi_exp
warning: function 'phi_exp' defined within script file 'xxx/phi_exp.m'
xs =
 -10   -9   -8   [...]
```

Then a script can be run from the command-line; for example:

```
$ octave --eval phi_exp.m --persist
# OR
$ octave phi_exp.m
```

Using Scilab

Scilab may be best obtained, on Arch Linux, from the AUR (ex: `yay -Sy scilab`) by selecting the `scilab-bin` option (relying then on prebuilt binaries); it can then be run with: `scilab`.

Note that the copy/paste behaviour is not consistent with the usual UNIX/X11 one, and that tabulation can be used for auto-completion.

Defining a Function Let's suppose we want to define $my\_func : x \to 2.x^2 + 1$.

For that, in Scilab's shell, enter:

```
--> function [y] = my_func(x)
  > y = 2*x^2+1
  > endfunction
```

Then:

```
--> my_func(5)
 ans  =
   51.
```

Plotting a Function Let's define the support of our function, here computed from 0 to 10 with 50 values: `my_xs = linspace(0, 10, 50)`. Then just execute `plot(my_xs, my_func)`.

We experienced rendering issues that prevented a proper display of plots.

Using Octave

Octave can be installed on Arch Linux with `pacman -Sy octave`; extra packages may be needed (ex: `octave-quaternion`, available in the AUR).

The command-line version can be run as `octave`. Typing `quit` at the prompt allows to exit.

The GUI version can be launched with `octave --force-gui`.

Defining a Function As already seen, so that it can operate on single values or arrays, a $\phi(x) = e^x/(1+e^x)$ function can be defined as:

```
function retval = phi (x)
  retval = exp(x)./(1+exp(x))
endfunction
```

Plotting a Function We consider first a function of a single variable.

Let `my_xs = 0:0.2:10` define the support / display range of our function; then:

```
my_ys = my_func(my_xs)
plot(my_xs, my_ys)
```

A key point is to understand that, for all plots (`plot`, `mesh`, `surf`, etc.), the last element to be specified is *not* the function to which the previous elements are to be applied, but directly the final values to plot.

Extra display settings can be added afterwards:

```
title ("This is my title")
ylabel ("My ordinate label")
xlabel ("My abscissa label")
grid on
```

This results in:

The image can be saved either by using the `Save As` GUI menu and typically selecting PNG, or directly from the console/scripts thanks to the following command: `print("my_plot.png", "-dpng")`.

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, use the contact address at the top of this document).

Ending Word

Hoping that these Ceylan-HOWTOs may be of help!

HOW-TO