Ceylan's HOWTO

HOW-TO

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

Creation date: Wednesday, November 17, 2021

Lastly updated: Sunday, November 21, 2021

Version: 0.0.1

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`).

Table of Contents

Build Tools

Organisation: Copyright (C) 2021-2021 Olivier Boudeville

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

Creation date: Saturday, November 20, 2021

Lastly updated: Sunday, November 21, 2021

Table of Contents

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

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

3D HOWTO

Organisation: Copyright (C) 2021-2021 Olivier Boudeville

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

Creation date: Saturday, November 20, 2021

Lastly updated: Sunday, November 21, 2021

Table of Contents

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.

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

Unreal Engine

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

Its licence is meant to induce costs only when making large-enough profits.

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.

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 without too much technical effort/constraint.

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`.

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").

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.

Assets They can be downloaded through the `Window -> Package Manager -> Packages:My Assets` menu option.

To access the assets provided by such packages, 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`.

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`)[1]; such directory generally contains at least 3 files:

- `asset`: the resource itself, renamed to that unique 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; one may use for that our blender-import.sh script.

3D Data

File Formats

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

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 and the related section of Godot.

The second best choice we see is Collada (`*.dae` files), an XML-based counterpart (open as well) to glTF.

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.

Samples

For:

- glTF, notably glTF 2.0; also a simple cube
- DAE; also a simple cube
- FBX

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

Modelling Software

Blender

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

Wings3D

Wings3D is a nice, Erlang-based, free software subdivision modeler.

OpenGL Corner

Hints

OpenGL allows the main program running on the CPU to communicate with typically a graphic card. As such most of the calls performed by user programs are asynchronous: through OpenGL they are triggered by the program 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.

Information

- FAQ for OpenGL and GLUT
- About OpenGL Performance

Erlang

Organisation: Copyright (C) 2021-2021 Olivier Boudeville
Contact: about (dash) howtos (at) esperide (dot) com
Creation date: Saturday, November 20, 2021
Lastly updated: Sunday, November 21, 2021

Table of Contents

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!

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

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 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).

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
- in addition to the module index mentioned in the **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
- for those who are interested in parse transforms (the Erlang way of doing metaprogramming), the section about The Abstract Format is esential (despite not being well known)
- to better understand the inner working of the VM: The Erlang Runtime System, a.k.a. "the BEAM book", by Erik Stenman

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

Docutils System Messages

system-message

ERROR/3 in `Erlang.rst`, line 164
Duplicate target name, cannot be used as a unique reference: "installation".