Contents

Building components

In this section, we cover all the components of the Civilization postgame generation pipeline one by one, just like in the [introductory section]. Before doing so, we first introduce the concept of a *namespace*.

Namespaces in viash

Once you start to make components with viash and combining them in larger scripts, workflows or pipelines you will quickly notice that some kind of grouping comes in handy:

1. Grouping helps in the bookkeeping related to functionality that is covered using components
2. Grouping helps in separating different concerns: different people may be interested in different types of components with a grouping mechanism each can focus on his their own domain.
3. Grouping helps in allowing to develop different sets of components in parallel and then later bringing those together in a larger project

We call a group of components a *namespace*.

Viash has a few ways to associate a namespace to a components:

1. ~~By means of a `namespace` attribute in the viash config~~
2. ~~By means of command line parameter when building an executable~~
3. By means of structuring the components properly and using the `viash ns` subcommand

Let us give an example of the first 2, option 3 will be used later in this section.

An example

We introduce a very simple component, one that only reports the release of an Alpine docker container, albeit the component could be used to `cat` the contents of other dockerized files as well:

`src/container_cat/config.vsh.yaml`:

```
functionality:
  name: container_cat
  arguments:
    - name: "file"
      type: string
      default: /etc/alpine-release
  resources:
    - type: executable
```

```
      path: cat
platforms:
  - type: docker
    id: docker1
    image: alpine:latest
  - type: docker
    id: docker2
    image: alpine:2.6
```

We introduce two Docker platforms that can be distinguished by id (`docker1` and `docker2`). Let us illustrate their use:

`docker1` platform:

```
> viash run src/container_cat/config.vsh.yaml -p docker1
3.13.0
```

`docker2` platform:

```
> viash run src/container_cat/config.vsh.yaml -p docker2
2.6.6
```

The component can be used to `cat` the contents of other files as well:

```
> viash run src/container_cat/config.vsh.yaml -p docker1 -- /etc/hosts
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2  b9ccb854b59a
```

Remark: Please note that we did not specify the argument as `type: file` because that would automount the *host*'s filesystem in the container (or at least attempt to). In this case, we effectively want to look inside the container.

A namespace: `container_tools`

Our `container_cat` example fits nicely in a collection of components to deal with containers and so we want to attach the namespace `container_tools` to it. Let's see how this can be done.

The example in a namespace

We take the example from above, but now store it in a directory hierarchy like this:

```
> tree src/container_tools
src/container_tools
```

```
└── container_cat
    └── config.vsh.yaml

1 directory, 1 file
```

The directory `container_tools` corresponds to the name of the namespace. Apart from this, there is no difference in how `container_cat` is defined.

We can now use the `viash ns` subcommand like this:

```
> viash ns build -n container_tools
Exporting src/container_tools/container_cat/ (container_tools) =docker1=> target/docker1/container_
Exporting src/container_tools/container_cat/ (container_tools) =docker2=> target/docker2/container_
```

We specify the name of the namespace using the `-n` parameter. In this case, there is only one component in this namespace, but it contains two platforms. The `viash ns` command *builds* a *target* for every platform it detects unless an optional `-p` is specified in the command above.

As a matter of fact, even the `-n` option can be omitted in which case *all* namespaces under `src` will be parsed.

This is a very effective way of keeping a collection of components under `src` grouped in namespaces. Different namespaces could be split across different directories or even source repositories and then combined on the level of `viash` by specifying the *target* directory (`target/` by default).

`civ6_save_renderer` namespace

Looking at the contents of `src/civ6_save_renderer`, we notice that `civ6_save_renderer` is a namespace that contains a number of components:

```
> tree src/civ6_save_renderer
src/civ6_save_renderer
├── combine_plots
│   ├── config.vsh.yaml
│   └── script.sh
├── convert_plot
│   ├── config.vsh.yaml
│   └── script.sh
├── parse_header
│   ├── config.vsh.yaml
│   └── script.sh
├── parse_map
│   ├── config.vsh.yaml
│   ├── helper.js
│   └── script.js
└── plot_map
    ├── config.vsh.yaml
    ├── helper.R
    └── script.R

5 directories, 12 files
```

We cover the components one by one in what follows and discuss any specificities that we encounter underway.

`parse_header`

Let us start with `parse_header`, it parses the headers of the save files.

`src/civ6_save_renderer/parse_header/config.vsh.yaml`:

```yaml
functionality:
  name: parse_header
  namespace: civ6_save_renderer
  description: "Extract game settings from a Civ6 save file as a yaml."
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--input"
      alternatives: [-i]
      type: file
      required: true
      default: "save.Civ6Save"
      must_exist: true
      description: "A Civ6 save file."
    - name: "--output"
      alternatives: [-o]
      type: file
      required: true
      default: "output.yaml"
      direction: output
      description: "Path to store the output YAML at."
  resources:
    - type: bash_script
      path: script.sh
platforms:
  - type: docker
    image: node
    docker:
      run:
        - cd /home/node && npm install civ6-save-parser
  - type: native
```

`src/civ6_save_renderer/parse_header/script.sh`:

```bash
#!/bin/bash

node /home/node/node_modules/civ6-save-parser/index.js "$par_input" --simple > "$par_output"
```

`parse_map`

The next component is `parse_map`. It is similar in nature that `parse_header` but this time we need a Javascript library to get the file parsed properly:

`src/civ6_save_renderer/parse_map/config.vsh.yaml`:

```yaml
functionality:
  name: parse_map
  namespace: civ6_save_renderer
  description: "Extract map information from a Civ6 save file as a tsv."
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--input"
      alternatives: [-i]
      type: file
      required: true
      default: "save.Civ6Save"
      must_exist: true
      description: "A Civ6 save file."
    - name: "--output"
      alternatives: [-o]
      type: file
      required: true
      default: "output.tsv"
      direction: output
      description: "Path to store the output TSV file at."
  resources:
    - type: javascript_script
      path: script.js
    - path: helper.js
platforms:
  - type: docker
    image: node
  - type: native
```

src/civ6_save_renderer/parse_map/script.js:

```javascript
// read helper libraries & functions
const fs = require("fs");
const helper = require(resources_dir + "/helper.js");

// read data from file
const json = helper.savetomap(fs.readFileSync(par["input"]));

// convert to tsv
const headers = Object.keys(json.tiles[0]);
const header = headers.join("\t") + "\n";
const lines = json.tiles.map(o => {
  return Object.values(o).map(b => JSON.stringify(b)).join("\t") + "\n";
});
const tsvLines = header + lines.join('')
```

```
// save to file
fs.writeFileSync(par["output"], tsvLines);
```

This last script uses the following helper script: `src/civ6_save_renderer/parse_map/helper.js` but it is a bit too long to represent here in this document.

The container to run in is an off-the-shelve `node` container that will be pulled automatically at first use (or when calling `---setup` on the executable).

This component is not very special in itself, but we would like to point out that the power of viash lies in making sure that not only `script.js` are passed to the container at runtime, but also the `helper.js` file that is used by `script.js`. This is all done seamlessly without a need for the user to understand what is happening under the hood.

Please note the ease of use of having `par["input"]` etc. automatically at your disposal coming from the CLI arguments specified with the component configuration.

Imagine you would have to do this manually using the Docker CLI?

`plot_map`

Based on the output from `parse_header` and `parse_map`, we can now generate a plot using `plot_map`. We have some R code to do this but the code in itself uses some R libraries that are not standard. This is again a perfect use-case for a containerized solution that is leveraged using viash!

`src/civ6_save_renderer/plot_map/config.vsh.yaml`:

```
functionality:
  name: plot_map
  namespace: civ6_save_renderer
  description: "Use the settings yaml and the map tsv to generate a plot (as PDF)."
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--yaml"
      alternatives: [-y]
      type: file
      required: true
      default: "header.yaml"
      must_exist: true
      description: "A YAML file containing civ6 game settings information."
    - name: "--tsv"
      alternatives: [-t]
```

```
      type: file
      required: true
      default: "map.tsv"
      must_exist: true
      description: "A TSV file containing civ6 map information."
    - name: "--output"
      alternatives: [-o]
      type: file
      required: true
      default: "output.pdf"
      direction: output
      description: "Path to store the output PDF file at."
  resources:
    - type: r_script
      path: script.R
    - path: helper.R
platforms:
  - type: docker
    image: rocker/tidyverse
    r:
      cran:
        - ggforce
        - yaml
        - bit64
        - ggnewscale
        - cowplot
      github:
        - rcannood/civ6saves
  - type: native
```

src/civ6_save_renderer/plot_map/script.R:

```
library(tidyverse)
library(cowplot)

source(paste0(resources_dir, "/helper.R"))

# par <- list(
#   yaml = "<...>/workspace/di/viash_workshop_1/data.yaml",
#   tsv = "<...>/workspace/di/viash_workshop_1/data.tsv"
# )

# read data
game_data <- read_header(par$yaml)
map_data <- read_map(par$tsv)

# make visualisation
g <- make_map_plot(game_data, map_data)

# save map to file
```

```
gleg <- cowplot::get_legend(g)
gnoleg <- g + theme(legend.position = "none")
gout <- cowplot::plot_grid(gnoleg, gleg, rel_widths = c(8, 1))
ggsave(par$output, gout, width = 24, height = 13)
```

Again there is a helper script `helper.R` that we do not completely render in this document but can easily be retrieved by looking at the sources.

This component takes 2 input files, a `yaml` file and a `tsv` file

The container for this component is based on `rocker/tidyverse` but there are some modifications that are applied. Additional R libraries are installed, 5 from the CRAN database, 1 from Github. This functionality covers a major reason to create a custom `Dockerfile` an thus container image: customizing a base container to suite ones needs. Adding the customizations using the viash configuration entails similar benefits to viash generating the command-line parsing code, namely standardization.

Remark: Please note that also here `par$yaml`, `par$tsv`, ... are automatically at your disposal and are passed from the wrapped (and containerized) executable. The `script.R` file even contains a (commented) code block that if uncommented allows one to develop and run the script (in or outside a container) without viash. Often times, component development will be done like this, using tools like RStudio or Jupyter notebooks on the native system and once the script is ready it is then converted to a (viash) component.

`convert_plot`

We covered the `convert_plot` component in the previous section and so will only quickly render the relevant source files here:

`src/civ6_save_renderer/convert_plot/config.vsh.yaml`:

```
functionality:
  name: convert_plot
  namespace: civ6_save_renderer
  description: Convert a plot from pdf to png.
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--input"
      alternatives: [-i]
      type: file
      required: true
      default: "input.pdf"
```

```
      must_exist: true
      description: "A PDF input file."
    - name: "--output"
      alternatives: [-o]
      type: file
      required: true
      default: "output.png"
      direction: output
      description: "Output path."
  resources:
    - type: bash_script
      path: script.sh
platforms:
  - type: docker
    image: dpokidov/imagemagick
  - type: native
```

src/civ6_save_renderer/convert_plot/script.sh:

```
#!/bin/bash

convert "$par_input" -flatten "$par_output"
```

`combine_plots`

We covered the `combine_plots` component in the previous section and so will only quickly render the relevant source files here:

src/civ6_save_renderer/combine_plots/config.vsh.yaml:

```
functionality:
  name: combine_plots
  namespace: civ6_save_renderer
  description: Combine multiple images into a movie using ffmpeg.
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--input"
      alternatives: [-i]
      type: file
      required: true
      default: "plot1.png:plot2.png"
      must_exist: true
      multiple: true
      description: A list of images.
    - name: "--output"
```

```
      alternatives: [-o]
      type: file
      required: true
      default: "output.webm"
      direction: output
      description: A path to output the movie to.
    - name: "--framerate"
      alternatives: [-f]
      type: integer
      default: 4
      description: Number of frames per second.
  resources:
    - type: bash_script
      path: script.sh
platforms:
  - type: docker
    image: jrottenberg/ffmpeg
  - type: native
```

src/civ6_save_renderer/combine_plots/script.sh:

```
#!/bin/bash

inputs=$(echo $par_input | tr ':' '|')
ffmpeg -framerate $par_framerate -i "concat:$inputs" -c:v libvpx-vp9 -pix_fmt yuva420p -y "$par_ou
```

Building the namespace

We can easily convert the full contents this namespace into executables using:

```
> viash ns build -n civ6_save_renderer
Exporting src/civ6_save_renderer/combine_plots/ (civ6_save_renderer) =docker=> target/docker/civ6_s
Exporting src/civ6_save_renderer/combine_plots/ (civ6_save_renderer) =native=> target/native/civ6_s
Exporting src/civ6_save_renderer/convert_plot/ (civ6_save_renderer) =docker=> target/docker/civ6_sa
Exporting src/civ6_save_renderer/convert_plot/ (civ6_save_renderer) =native=> target/native/civ6_sa
Exporting src/civ6_save_renderer/parse_header/ (civ6_save_renderer) =docker=> target/docker/civ6_sa
Exporting src/civ6_save_renderer/parse_header/ (civ6_save_renderer) =native=> target/native/civ6_sa
Exporting src/civ6_save_renderer/parse_map/ (civ6_save_renderer) =docker=> target/docker/civ6_save_
Exporting src/civ6_save_renderer/parse_map/ (civ6_save_renderer) =native=> target/native/civ6_save_
Exporting src/civ6_save_renderer/plot_map/ (civ6_save_renderer) =docker=> target/docker/civ6_save_m
Exporting src/civ6_save_renderer/plot_map/ (civ6_save_renderer) =native=> target/native/civ6_save_m
```

Remark: Please note that both the `docker` platform as well as the `native` platform are taken into account. Because most people will not have the necessary tools for running the different steps, we will not build the executables for the `native` platform:

```
> rm -r target
```

And then:

```
> viash ns build -n civ6_save_renderer -p docker
Exporting src/civ6_save_renderer/combine_plots/ (civ6_save_renderer) =docker=> target/docker/civ6_s
Exporting src/civ6_save_renderer/convert_plot/ (civ6_save_renderer) =docker=> target/docker/civ6_sa
Exporting src/civ6_save_renderer/parse_header/ (civ6_save_renderer) =docker=> target/docker/civ6_sa
Exporting src/civ6_save_renderer/parse_map/ (civ6_save_renderer) =docker=> target/docker/civ6_save_
Exporting src/civ6_save_renderer/plot_map/ (civ6_save_renderer) =docker=> target/docker/civ6_save_m
```

This is what the `target` directory looks like now:

```
> tree target/
target/
└── docker
    └── civ6_save_renderer
        ├── combine_plots
        │   └── combine_plots
        ├── convert_plot
        │   └── convert_plot
```

```
        ├── parse_header
        │   └── parse_header
        ├── parse_map
        │   ├── helper.js
        │   └── parse_map
        └── plot_map
            ├── helper.R
            └── plot_map

7 directories, 7 files
```

Please notice a few things:

- Every components has its own directory under `target/<platform>/<namespace>/`
- The `script.R`, `script.sh`, … files are contained in the respective executables, helper files are passed at runtime.
- Every *target* component directory contains a `viash.yaml` file which contains necessary (meta) information for reproducing the component

Using the respective (containerized) tools is now as easy as, for instance,

```
> target/docker/civ6_save_renderer/parse_header/parse_header -i ../data/AutoSave_0158.Civ6Save -o /
```

`/tmp/output.yaml`:

```
{
  ACTORS: [
    {
      START_ACTOR: 4159575459,
      ACTOR_NAME: 'CIVILIZATION_FREE_CITIES',
      ACTOR_TYPE: 'CIVILIZATION_LEVEL_FREE_CITIES',
      ACTOR_AI_HUMAN: 1,
      LEADER_NAME: 'LEADER_FREE_CITIES'
    },
    {
... (cut) ...
```