Contents

Creating components with viash

Data Intuitive Tuesday - January 26, 2021

With the information from the previous section, we will wrap two components from the Civ6 use case in more detail (`convert_plot` and `plot_map`). Let's start with the easiest of the two.

`convert_plot` - a simple bash script

If you recall, `convert_plot` is responsible to converting the game map pdf to a png using [ImageMagick]. We will wrap this component first it is a perfect example of one of the most common type of viash components: wrapping a tool in a simple bash script.

Converting pdfs manually

Installing [ImageMagick] on a Debian-based system very easy.

```
sudo apt-get install imagemagick
```

Converting a pdf to a png is a simple one-line command.

```
#!/bin/bash

par_input=data/AutoSave_0159.pdf
par_output=data/AutoSave_0159.png

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

Additional arguments can be provided, but are not required since [ImageMagick] is pretty good at getting the defaults right.

In order to wrap this script as a viash component, we need to write a viash config file. Let's start by describing the functionality in a yaml file.

Step 1: Functionality

The functionality describes what the component is expected to do, with which arguments, and with which resources (files). For now, we provided a description on what the component does and who wrote it.

```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: # to do, see step 2
  resources: # to do, see step 3
```

Step 2: Functionality arguments

Defining the arguments is also relatively simple, as there is only a single 'input' and a single 'output' file.

```yaml
  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."
```

Arguments have many optional fields which allow to fine tweak their behaviour.

- `alternatives: [-i]` - Alternative (shorter) flags to pass parameter values with.
- `type: file` - The value for this argument is either a file or a directory.
- `required: true` - A parameter value needs to be passed when the component is invoked.
- `default: "input.pdf"` - Sets a default value. This is irrelevant when required is true, but the value will still be used when rendering the CLI help page.

- `must_exist: true` - This file needs to exist when the component is invoked.
- `direction: output` - This file is an output file, not an input file. If this value is not specified, then `"input"` is assumed by default.
- `description: "..."` - A human-readable description of this argument. Note that multiline strings can be used, if desired.

Step 3: Bash script

There is only a single Bash script called `script.sh`. We can specify its existence as follows:

```
  resources:
    - type: bash_script
      path: script.sh
```

The file `script.sh` is not so much different from the example we gave above. The only difference is that the `par_input` and `par_output` variables are surrounded by some comments. How can this script work properly if the values of `par_input` and `par_output` are always the same?

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

```
#!/bin/bash

## VIASH START
par_input=input.pdf
par_output=output.png
## VIASH END

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

Well, the code block between `## VIASH START` and `## VIASH END` is entirely optional. In fact, viash will not even look at its content. When you run a viash component, the code block above will be replaced with some code to create a command-line interface through which to pass the parameters.

Why is this code block there in the first place? It is simply a placeholder where you can store some testing parameters while you are writing the script (so you can run `./script.sh` while writing it).

Side note: This syntax is similar but slightly different depending on the scripting language used.

Step 4: Define platform(s)

Finally, we need to specify what the system requirements are for viash to be able to run this component. If nothing is specified, the script will automatically be run on the host system natively. By going one step further, we can specify how the component can run inside a docker container. Luckily, there is already a Docker image available on Docker

Hub for running [ImageMagick], namely dpokidov/imagemagick. We can thus specify the platforms as follows:

```
platforms:
  - type: docker
    image: dpokidov/imagemagick
  - type: native
```

Note that had this not been the case, the solution would have been to use a standard container (e.g. Ubuntu) and install ImageMagick manually:

```
platforms:
  - type: native
  - type: docker
    image: ubuntu
    setup:
      - type: apt
        packages: imagemagick
```

Luckily, we didn't need to resort to such drastic measures.

Step 5: Final config

This is the final viash config for this component in its entirety.

`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: nextflow
    image: dpokidov/imagemagick
  - type: native
```

By notation, the file extension of a viash config file should be `.vsh.yaml`, otherwise it will not be detected by viash.

Step 6: Building the executable

Building an executable can be done just like before. We assume [ImageMagick] is not installed on the local system and thus build the Docker version:

```
> viash build src/civ6_save_renderer/convert_plot/config.vsh.yaml -o bin -p docker
```

We specify the `docker` platform explicitly although that is not really necessary because of the order of the platforms in the viash config. The resulting script is stored under `bin` relative to the current working directory.

We ask the generated executable to run the necessary setup. In this case, it means *pulling* the appropriate docker image from Docker Hub.

```
> bin/convert_plot ---setup
> docker pull dpokidov/imagemagick
Using default tag: latest
latest: Pulling from dpokidov/imagemagick
Digest: sha256:6749db04ffa5eac1cbe77566af02463f040028fef525b767dc98e06023e6cdf8
Status: Image is up to date for dpokidov/imagemagick:latest
docker.io/dpokidov/imagemagick:latest
```

We can display the help page of the component as follows:

```
> bin/convert_plot -h
Convert a plot from pdf to png.

Options:
    -i file, --input=file
        type: file, required parameter, default: input.pdf
        A PDF input file.

    -o file, --output=file
        type: file, required parameter, default: output.png
        Output path.
```

Step 7: Running the executable

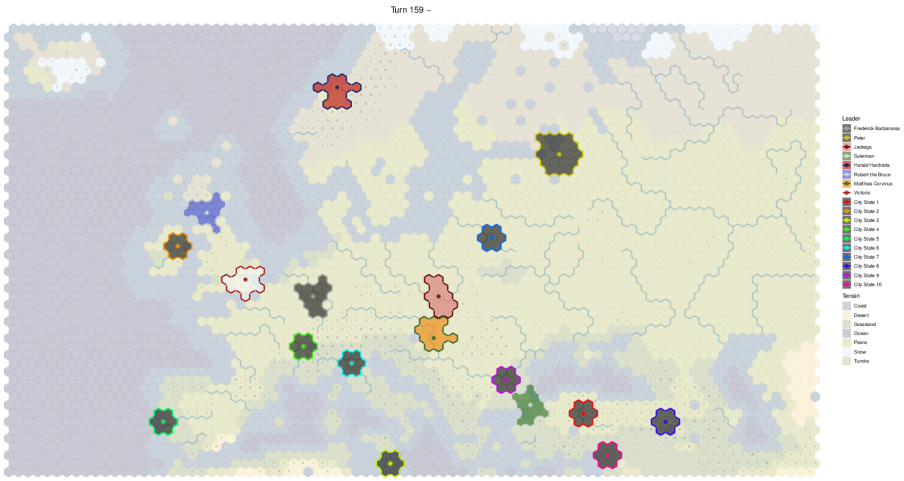
Now that everything is up and running, we can start converting images. Let us first generate a simple PDF file with the help of viash. We start by preparing a directory to store the data:

```
> mkdir -p output/
```

We can generate a png file from a pdf file as follows.

```
> bin/convert_plot -i data/AutoSave_0159.pdf -o data/AutoSave_0159.png
```

`output/AutoSave_0159.png`:



Note on file management when running Docker images

Please note that in the above example, the input file and output file reside on the host image while the conversion process is running inside a Docker container. If we would want to achieve this without viash, we would need something like this:

```
docker run -i -v `pwd`:/mount dpokidov/ImageMagick /mount/data/input.pdf -flatten /mount/data/outpu
```

This requires some mental bookkeeping to understand the difference between the host's file system and the one inside the container. It also requires one to know how the container's commands are parsed. In this case the `convert` command from [ImageMagick] is automatically called with the options we provide. But that may be different for every container and depends on the contents of the `Dockerfile`.

Also, while the above explicit `docker` command achieves our aim, it does not fully cover the use-case that we tackle using viash. For a correct comparison, we would have to run our custom script in the container. But then, we would have to make a few updates:

1. Include command-line argument parsing in the `script.sh` file so that we can provide input and output parameters to it.

2. *Install* the modified `script.sh` file inside the container, or somehow *mount* the location of the executable inside the container such that it can be found.

In other words:

> Using viash all this is greatly simplified and wrapped in one executable, command-line parsing comes for free.

`plot_map` - a more complicated R script

This component is a great example of a script which is a bit more complicated. Currently, viash supports wrapping the following scripting languages: Bash, R, Python, JavaScript, and Scala. Of course, if your favourite scripting language is not supported (yet), you can simply run it from a Bash script.

The `plot_map` component takes a `yaml` and a `tsv` file as input and outputs a map view of the information as a pdf. Note that the output of this component is actually the input of the previously mentioned component, `convert_plot`. That's okay; you can develop viash components in any order you like!

Let's start by defining the functionality-part of the viash config.

Step 1: Functionality

The functionality metadata is largely the same when compared to that of the previous component.

```
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
```

One notable difference is that the code is split up into two files: the main script `script.R` and a helper file `helper.R`. By splitting of a portion of the code into a second file, the main script becomes a bit more readable.

Step 2: R script

This is the contents of the R script. The comments provide some sense of how this script works:

src/civ6_save_renderer/plot_map/script.R:

```
# load helper functions 'read_header()', 'read_map()', and 'make_map_plot()'
source(paste0(resources_dir, "/helper.R"))

# load libraries
library(tidyverse)
library(cowplot)

## VIASH START
par <- list(
  yaml = "data.yaml",
  tsv = "data.tsv",
  output = "output.pdf"
)
## VIASH END

# 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)
```

For the sake of brevity, we will not show the contents of the `helper.R` file, as all it contains are three functions which do some magic hocus pocus transformations of the yaml and the tsv file in order to render a [ggplot2] visualisation.

Step 3: Define platform(s)

In order to run this component natively, we need a working R installation, combined with a specific set of R packages (listed a the top of the different R files, e.g. `library(cowplot)`).

We can start from the `rocker/tidyverse:4.0.3` container, which already contains all of the tidyverse packages, but some extra dependencies need to be installed.

```
platforms:
  - type: docker
    image: "rocker/tidyverse:4.0.3"
    setup:
      - type: r
        cran:
          - ggforce
          - yaml
          - bit64
          - ggnewscale
          - cowplot
        github:
          - rcannood/civ6saves
  - type: native
```

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: extending a base container to suit ones needs. Adding the extra software using the viash configuration entails similar benefits to viash generating the command-line parsing code, namely standardisation.

Step 4: Building and running the component

Building an executable and running it is very similar to before. Using the `--setup` flag, we can let viash run the `---setup` command after building the executable without much effort. Unfortunately, installing R packages can take a lot of time, so this command might take a while to run.

```
> viash build src/civ6_save_renderer/plot_map/config.vsh.yaml -o bin -p docker --setup > /dev/null
```

This is the corresponding generated help page.

```
> bin/plot_map -h
Use the settings yaml and the map tsv to generate a plot (as PDF).

Options:
    -y file, --yaml=file
        type: file, required parameter, default: header.yaml
        A YAML file containing civ6 game settings information.

    -t file, --tsv=file
        type: file, required parameter, default: map.tsv
        A TSV file containing civ6 map information.

    -o file, --output=file
        type: file, required parameter, default: output.pdf
        Path to store the output PDF file at.
```

Finally, we can run the component as follows.

```
> bin/plot_map \
+   --yaml data/AutoSave_0159.yaml \
+   --tsv data/AutoSave_0159.tsv \
+   --output data/AutoSave_0159.pdf
```

Conclusion

In this part of the tutorial, we learned how to wrap Bash and R scripts from scratch.

For more information on the specifications of viash config files, visit the documentation on viash configs.

In the next section, we will take a look at how to build a collection of viash components.