Contents

Creating a simple pipeline in Bash

In this section, we will cover how to build all the Civ6 postgame components and chain them together in a first rudimentary pipeline written in Bash. Before doing so, we first introduce the concept of a *namespace*.

Namespaces

Once you start to develop a number of viash components, grouping them (hierarchically) allows to improve maintenance of the components as it allows for separation of concern. In addition, multiple developers could group on different sets of components in parallel and later bring them together in a larger project. We call a group of components a *namespace*.

You can assign a namespace to a component by setting the `namespace` attribute in a viash config:

```
functionality:
  name: some_component
  namespace: my_namespace
```

Building a namespace

Alternatively, the namespace can be automatically inferred by structuring the components hierarchically and using the `viash ns` (read: viash namespace) command. You may have noticed that the components in the `src` directory of this repository already are structured in this manner:

```
> tree src
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
├── markdown_tools
│   ├── cat_format
│   │   ├── config.vsh.yaml
│   │   └── script.sh
│   └── render_table
│       ├── config.vsh.yaml
│       └── script.R
└── simple_pipeline.sh

9 directories, 17 files
```

With `viash ns build` you can build all the components in a namespace. If we only wish to build the Civ6 postgame components, we can specify the name of the namespace using the `-n` parameter.

```
> 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) =nextflow=> target/nextflow/c
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) =nextflow=> target/nextflow/civ
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) =nextflow=> target/nextflow/civ
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) =nextflow=> target/nextflow/civ6_s
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
Exporting src/civ6_save_renderer/plot_map/ (civ6_save_renderer) =nextflow=> target/nextflow/civ6_sa
```

In this case, there are five components in this namespace, but multiple platforms (native, docker, nextflow) for each of them. The `viash ns` command *builds* a *target* for every platform it detects unless an optional `-p` is specified in the command above. By omitting the `-n`, viash will build

all namespaces in the `src` folder. The `viash ns build` command 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.

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
+ viash ns build -n civ6_save_renderer -p docker --setup > /dev/null
```

Since we have to run the *setup* for the containers that are not just available on Docker Hub, we provide an additional `--setup` flag to let viash take care of this for us.

Manually running executables

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.

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

```
> target/docker/civ6_save_renderer/parse_header/parse_header -i data/AutoSave_0159.Civ6Save -o data
```

`data/AutoSave_0159.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) ...
```

A first pipeline in Bash

A small dataset with only a few steps from a game are stored under `data/`. We will use that as a source for the pipeline.

With the following script:

`src/simple_pipeline.sh`:

```
#!/bin/bash

input_dir="data"
output_dir="output"
CIV6="target/docker/civ6_save_renderer"

mkdir -p "$output_dir"

# iterate over every Civ6Save file
for save_file in $input_dir/*.Civ6Save; do
  file_basename=$(basename $save_file)

  echo ">>>>>>> parse header '$save_file'"
  yaml_file="$output_dir/${file_basename/Civ6Save/yaml}"
  $CIV6/parse_header/parse_header -i "$save_file" -o "$yaml_file" 2&>1 > /dev/null

  echo ">>>>>>> parse map '$save_file'"
  tsv_file="$output_dir/${file_basename/Civ6Save/tsv}"
  $CIV6/parse_map/parse_map -i "$save_file" -o "$tsv_file" 2&>1 > /dev/null

  echo ">>>>>>> plot map '$save_file'"
  pdf_file="$output_dir/${file_basename/Civ6Save/pdf}"
  $CIV6/plot_map/plot_map -y "$yaml_file" -t "$tsv_file" -o "$pdf_file" 2&>1 > /dev/null

  echo ">>>>>>> convert plot '$save_file'"
  png_file="$output_dir/${file_basename/Civ6Save/png}"
  $CIV6/convert_plot/convert_plot -i "$pdf_file" -o "$png_file" 2&>1 > /dev/null
done

echo ">>>>>>>combine plots"
```

```
png_inputs=`find "$output_dir" -name "*.png" | tr '\n' ':'`
$CIV6/combine_plots/combine_plots -i "$png_inputs" -o "$output_dir/movie.webm" --framerate 1 2&>1 >

echo ">>>>>>>DONE"
```

Running it yields the following results.

```
> src/simple_pipeline.sh
>>>>>>> parse header 'data/AutoSave_0158.Civ6Save'
>>>>>>> parse map 'data/AutoSave_0158.Civ6Save'
>>>>>>> plot map 'data/AutoSave_0158.Civ6Save'
>>>>>>> convert plot 'data/AutoSave_0158.Civ6Save'
>>>>>>> parse header 'data/AutoSave_0159.Civ6Save'
>>>>>>> parse map 'data/AutoSave_0159.Civ6Save'
>>>>>>> plot map 'data/AutoSave_0159.Civ6Save'
>>>>>>> convert plot 'data/AutoSave_0159.Civ6Save'
>>>>>>> parse header 'data/AutoSave_0160.Civ6Save'
>>>>>>> parse map 'data/AutoSave_0160.Civ6Save'
>>>>>>> plot map 'data/AutoSave_0160.Civ6Save'
>>>>>>> convert plot 'data/AutoSave_0160.Civ6Save'
>>>>>>> parse header 'data/AutoSave_0161.Civ6Save'
>>>>>>> parse map 'data/AutoSave_0161.Civ6Save'
>>>>>>> plot map 'data/AutoSave_0161.Civ6Save'
>>>>>>> convert plot 'data/AutoSave_0161.Civ6Save'
>>>>>>> parse header 'data/AutoSave_0162.Civ6Save'
>>>>>>> parse map 'data/AutoSave_0162.Civ6Save'
>>>>>>> plot map 'data/AutoSave_0162.Civ6Save'
>>>>>>> convert plot 'data/AutoSave_0162.Civ6Save'
>>>>>>>combine plots
>>>>>>>DONE
```

Conclusions

While this bit of Bash scripting made this pipeline easy to write, there are some clear issues with it.

- All the results are produced sequentially. This strongly limits scalability as the number of samples in the datasets increases.
- A lack of parameterisation. As `input_dir` and `output_dir` are fixed, you need to modify this script every time you want to run it on a new dataset.
- No caching of results. Running the script twice will result in computing the results twice, even if they are already available.

These issues can all be fixed with some more Bash scripting (and some even by viash!), we'd be reinventing the wheel as this is all covered by Nextflow.

In the next section, we will review some best practices when writing new components with viash, before moving on to part 2 (hint: Nextflow!).