Game Character Hub Toolkit

gchlib: a command line tool to manage GCH resources

Sebastien Bini

Contents

1 Introduction **4**
- 1.1 Presentation of Game Character Hub . . . 4
- 1.2 Game Character Hub - Portfolio Edition items . . . 4

2 Getting started **6**
- 2.1 Launching gchlib for the first time . . . 6
- 2.2 Running the first commands . . . 6
- 2.3 Understanding gchlib workflow . . . 8
  - 2.3.1 gchlib new/set command summary . . . 8
  - 2.3.2 Combining gchlib with external tools . . . 8

3 gchlib objects **10**
- 3.1 Description of gchlib objects . . . 10
- 3.2 Combining objects to define items and packs . . . 11

4 gchlib commands overview **15**
- 4.1 Initialize command . . . 15
- 4.2 Add/Remove commands . . . 15
- 4.3 List command . . . 16
- 4.4 New/Set commands . . . 16
  - 4.4.1 New/Set spritesheet command . . . 17
  - 4.4.2 New/Set portfolio command . . . 17
  - 4.4.3 New/Set layer command . . . 18
  - 4.4.4 New/Set colorsheet command . . . 18
  - 4.4.5 New colorsheetgroup command . . . 20
  - 4.4.6 New/Set item command . . . 20
  - 4.4.7 New/Set pack command . . . 21
  - 4.4.8 New/Set license command . . . 21
- 4.5 Link command . . . 22
  - 4.5.1 Link layer command . . . 22
  - 4.5.2 Link colorsheet command . . . 24
  - 4.5.3 Link item command . . . 26
- 4.6 Make command . . . 28
  - 4.6.1 Make item command . . . 28
  - 4.6.2 Make pack command . . . 28
- 4.7 Dump command . . . 28

4.7.1 Dump item command 28
4.7.2 Dump pack command 28
4.8 Execute command 28

5 Appendix: grammar of json objects **29**
5.1 Json grammar of the Repository 29
5.2 Json grammar of object type "layer" 30
5.3 Json grammar of object type "spritesheet" 31
5.4 Json grammar of object type "portfolio" 31
5.5 Json grammar of object type "colorsheet" 31
5.6 Json grammar of object type "item" 32
5.7 Json grammar of object type "animation" 35
5.8 Json grammar of object type "generator" 35
5.9 Json grammar of object type "license" 37
5.10 Json grammar of object type "frameMappingProcess" 37
5.11 Json grammar of object type "pack" 38
5.12 Json grammar of object type "colorsheetgroup" 38

Chapter 1

Introduction

1.1 Presentation of Game Character Hub

Game Character Hub (GCH) is an art program that is made to help the creation of various characters for various game engines. In order to ease the character creation part, GCH comes with a complete *item library*. This library is basically made of PNG images that represent character parts. For instance you may have library items that represent various pieces of hair, various body parts, various outfits, etc... Those items can be freely mixed together via a layer system to create new unique characters. These characters can then be animated by GCH's animation player, reworked with GCH's various tools and finally be exported as PNG files to used in your game engine.

As GCH strives to remain game-engine agnostic, everything related to the game engine is configurable via GCH's user interface. For instance, animations can be configured so it supports your game engine. Even the character frame layout can be rearranged at export time to match your game engine's.

The item library can also be configurable, meaning that you can freely edit items, install new item packs, or import your own resources. In GCH 2.0, you could import your own resources into GCH's *item library* via the user interface. Most items were made of a single PNG file. A few items were more convoluted and would extend into several layers, which would require several input PNG files to create them. It was then feasible at a time to import your own PNG files into the *library* via a bunch of windows.

1.2 Game Character Hub - Portfolio Edition items

In Game Character Hub - Portfolio Edition (GCH-PE), *library* items got many new features (extend to several spritesheets, have several colorsheets, are automatically correctly positioned on a new character, ...), which on the other hand made them more complex to manage. While items used to be a bunch of PNG images, now in GCH-PE:

- each item can extend to several spritesheets.
- for each spritesheet it covers, the item can extend to several layers.

- each spritesheet has properties among: its size, its animation, its flavor, etc...
- each layer has properties among: its position, its opacity, its blending, etc...
- each item can have several color variations (handled by colorsheets).
- each colorsheet may recolorize any layer of any spritesheet defined in the item.
- each colorsheet can be uniquely flagged so that it can be synchronized with colorsheets of other library items.

This increase of complexity made it quite hard to design a user-friendly interface that would allow end users to import their own resources like they used to. Hence we decided that a user interface would no longer be approriate. Instead we provide you with a new tool: *gchlib*. This is a command-line tool that allows you to import items with a level of flexibility and scriptabiblity that a user interface would have never permitted.

Chapter 2

Getting started

2.1 Launching gchlib for the first time

gchlib is a command-line tool and, as such, must be executed from a command prompt, or a terminal. First, make sure that gchlib is in your PATH environment variable. Then, from the command prompt, go to the directory where your resources to import are located (for instance if your PNG files to import are located both at "XXX/My pack/Male" and "XXX/My pack/Female" and their subfolders, you should go to "XXX/My pack"). Going to a directoy is usually achieved by the command "**cd** *directory*" where *directory* is the directory location where you want to go.

Once you are in the directory, simply run *gchlib init*. It should run without any issue. This prepares the directory to hold GCH resources. Technically, it simply created a .gchlib file. This .gchlib is called the *repository*. In order to import your resources, you will have to create several gchlib files. Each will give some details on how to import your resources. Every gchlib file related to your resources will be referenced in the *repository*.

2.2 Running the first commands

With gchlib, you will be able to:

- combine many PNG images into a single GCH library item.
- combine many GCH library items into a GCH library pack that can be later installed via GCH.

Note: gchlib will simply create GCH item library files, but it will not import them into GCH. Note that importing items is relatively easy. Simply move the items gchlib generated into GCH item library folder, and GCH will recognize them. If you created a pack, then you have to move the generated .gchpack file into the *itempacks* folder of GCH installation directory. Upon launch, GCH will offer you to install the new pack(s).

For simplicity, in this section, we will focus on the first use case. Once you are familiar with gchlib, you will notice how easy it is to generate a pack given the items.

Let's get started. gchlib expects you to define how your future library item is constituted. This implies that you must:

- define its spritesheets and their properties.
- define its layers and their properties.
- assign a PNG file to each layer.

We could also define colorsheets, animations, a license, and many other meta data. But the mere requisites are simply these three steps.

See how to import an item that covers a single spritesheet and has a single layer:

```
gchlib new spritesheet walk -p spr.walk.json -flv walk -gridw 3 -gridh 4 -framew
48 -frameh 48
gchlib new portfolio mypack_portfolio -p ptf.json -spr walk
gchlib new layer body -p lyr.body.json -spr walk
gchlib new item item_body -p item.body.json -ptf mypack_portfolio
gchlib link layer item_body -lyr body -src "Male/Body.png"
gchlib make item -item item_body
```

The first line defines a spritesheet, which has a grid size of 3x4 (options gridw/gridh) and a frame size of 48x48 (options framew/frameh). Its flavor is "walk" (option flv). After this first line, gchlib will have saved a file named "spr.walk.json" (option p) which will contain the spritesheet properties. This file will be referenced in the *repository*under the name "walk" (argument after "new spritesheet").

The second line defines a portfolio. A portfolio is just a container of spritesheets. Here it creates a portfolio referenced as "mypack_portfolio", saved into "ptf.json", that contains the spritesheet "walk" (option spr).

The third line defines a layer. This layer belongs to the spritesheet "walk", which is the spritesheet we just defined. Likewise gchlib will store the layer properties into a file named "lyr.body.json", and the layer will be referenced as "body".

The forth line defines a new *item*. These *item*s can later be exported into actual GCH library items and moved directly into the library. The item uses the portfolio "mypack_portfolio" that we defined above (option ptf). At this time, the item is completely blank. We will populate it via the "link" command.

The fifth line links a PNG file (option src) to a layer (option lyr). This is the image that will be used to build the item when it is exported. Note that the layer "body" is known to the item: the item uses the portfolio "mypack_portfolio", which contains the spritesheet "walk", which itself contains the layer "body".

The last line actually creates the item: thanks to all the files and the associations we made above, gchlib manages to build the item and export it as a gpfi file. The output file can simply be moved into GCH item library folder and is ready to be used.

You might think that six commands just to define and export an item is too long. Note that the intermediary objects we defined when creating the first item (namely: the layer, the spritesheet, and the portfolio) can be reused for any other items. Hence this works as well:

```
gchlib new item item_largebody -p item.largebody.json -ptf mypack_portfolio
gchlib link layer item_largebody -lyr body -src "Male/LargeBody.png"
gchlib make item -item item_body -item item_largebody
```

This creates a new *item* that has the same format as the previous one, and that is linked to another PNG file. The last command will export both *item*s into gpfi files.

2.3 Understanding gchlib workflow

In the last section we had a glimpse of some gchlib commands. Let us now describe more in details the usual gchlib workflow and how to use it properly.

2.3.1 gchlib new/set command summary

gchlib works by defining intermediary objects (e.g. layers, spritesheets, colorsheets, ...) and their relationships (e.g. which layers are contained in which spritesheet, ...). Each object is stored in a file, whose location is registered in the *repository*(i.e. the .gchlib file).

It is possible to distinguish two main categories of gchlib objects. On one hand, there are objects that help define the final *item format* (layers, spritesheets, portfolios, animations, colorsheets, etc...) and that can be reused across several *item*s, as they do not contain any resource-specific information (i.e. they do not inform about any PNG file location to use to export the items). On the other hand, there are objects that are specific to an *item* (e.g. objects of type item or pack), as they define which PNG files to use to fill the *item*'s layers and colorsheets. There are for instance as many objects of type *item* as there are actual gpfi items to create. The first category can be seen as the logical definition of the *item format*. The second category as the physical definition of the *item*s.

gchlib allows to create these objects through various "new" commands. The general syntax is:

```
gchlib new <object-type> <object-reference> -p <object-file-location> [options...]
```

As seen in the examples, this creates a new file named ¡object-file-location¿. Its location is registered in the *repository*as ¡object-reference¿. It is also possible to edit the object properties even after it has been created via the "new" command. This is achieved via the "set" command:

```
gchlib set <object-type> <object-reference> [options...]
```

The complete command syntax is detailed in the next chapters.

2.3.2 Combining gchlib with external tools

You are encouraged to combine gchlib with other tools. In fact gchlib was designed such that it is easy (and even recommended) to mix it with scripting languages or any other text-

processing tools. You do not intend you to type every gchlib command by hand if you aim to create a whole item pack.

All the files created by gchlib are JSON files (even the *repository*). **You can freely edit any of them by hand, instead of using gchlib commands.** gchlib will not notice. This implies that, if for some reason you don't want to use gchlib new/set commands, you can use external tooling to create/edit gchlib files. You may want for instance to use your own *perl* or *python* scripts that will build gchlib JSON files from scratch (but this is not the recommended way, see below).

Beware though, gchlib makes many consistency checks and ensures that all the relationships between each object are consistent. Consequently, using external tools and bypassing gchlib exposes you to potential inconsistencies (for instance, references to files that no longer exist). If you stick with gchlib and never edit a file by hand, then you are ensured that your data will remain consistent.

Hence if you need to use external tools, it is not advised to directly generate gchlib JSON files. The external tools should generate gchlib commands instead and play them. By doing so, you will benefit from gchlib's consistency checks and minimize the risks of having inconsistent data.

Finally, it is also recommended to use a source code manager (like *git* or *mercurial*) along with gchlib. gchlib only uses JSON text files which makes it very convenient to use with such tools. This will help track changes between different versions of an item pack for instance. You will also feel more confident experimenting with gchlib if you know you cannot lose all your work by mistake.

Chapter 3

gchlib objects

3.1 Description of gchlib objects

Recap: As we have seen in the introduction chapter, gchlib work with JSON files. These JSON files help define the structure of each future library item and its contents. They can also define complete library pack. All these files are referenced in a single JSON file named the *repository*.

gchlib files are all classified by *object type*, which is basically the type of data the file defines. All JSON files of the same type will use the same JSON grammar. This is actually how gchlib manages to parse the JSON files: since it knows the object type of each file (thanks to the *repository*), it knows how to read them.

There are 11 object types handled by gchlib:

spritesheet Defines the structure of a spritesheet (grid size, frame size, etc...). It contains all the information about its dimensions and other meta data (such as the spritesheet name).

layer Defines layer properties (name, opacity, etc...).

portfolio Defines the structure of an item, i.e. a portfolio. Contains the list of spritesheets a given can support.

colorsheet Defines colorsheet properties. Colorsheets are used to define item color variations.

colorsheetgroup Defines a colorsheet group. This can be used to link colorsheets across different items.

item Defines a GCH library item, from its structure (portfolio) to its contents (PNG resources).

pack Defines a GCH library pack, from its properties to its files.

license Defines a license that can be attached to an item or a pack.

fmp Defines a frame mapping process.

generator Defines a GCH character generator.

animation Defines a GCH animation that can be played via GCH's animation player.

Most of these files can be built via gchlib new/set commands (see next chapter). It is not the case for the last 3 object types, as gchlib does not provide any new/set commands for them. If you want to create them (which is not mandatory to make a functional library item or pack), you will either have to write the files by hand, either use GCH user interface. Indeed, you can create or edit frame mapping processes, generators and animations through GCH windows. GCH offers the option to export them in JSON format. Once you have the JSON files handy, you can register them in the *repository*via the "add" command (see next chapter).

3.2 Combining objects to define items and packs

In this section, we will describe the relationships between each object, and how they should interact together to define library items and packs.

It is pretty common that a file relies on something that is defined on another file. In such cases, gchlib does not "hard copy" the contents of one file to the other. Instead, when this dependency happens, the file simply contains the *reference* of the other file it needs. gchlib will be able to retrieve the file given its reference thanks to the *repository*. This system offers more flexibility: if you need to edit a file, you only need to edit once.

Here is the relationships between each items: **spritesheet:** Spritesheet objects may depend on an **animation** object, which would be the animation that would be played in GCH's animation player. This means that a spritesheet object can contain a reference to an **animation** object.

layer: Spritesheets are made of those. A layer can only be into a single spritesheet. As such, a layer contains a reference to a spritesheet object, which is the **spritesheet** the layer is in.

portfolio: Portfolios are made of spritesheets. Unlike layers that can only into a unique spritesheet, spritesheets are free to be shared in any number of portfolios. Hence, each portfolio contain a list of references to **spritesheet** objects.

colorsheet: Colorsheets can be grouped together. This helps define a color variation for a character across different GCH items. As a given colorsheet may only be in a single group, each colorsheet may have a single reference to a **colorsheetgroup** file.

item: This defines a GCH item and as such, it must know about its structure. Hence every item holds a reference to a **portfolio**. Items can also have color variations, which implies they have references to some **colorsheet** files. And finally, items must know which PNG file to use for which layer, so they also contain references to the **layer**s objects they use. Optionally, an item may have a license, and hence may hold a reference to a **license** object.

pack: A pack defines a GCH library file that will be used by GCH to install a part of the item library. A pack can installs GCH items (and hence has references to the **item**s to install), but not only. A pack can also install **frame mapping process**es, **animation**s, **portfolio**s

and character **generator**s (and contains references to any number of any of these). Finally a pack may also have a **license**, and so it may contain a reference to one.

colorsheetgroups, licenses, frameMappingProcesses, generators, animations: These does not have external dependencies.

Now that we have established the relationships between each object, we can go through all the required steps to create an item:

1. As the spritesheets are a core component, and that they do not depend on other objects (except for animations but they are optional), it is a good idea to start with them. First we start by defining all the spritesheets that are required for our future items or pack. Keep in mind that a spritesheet simply define a format and tells nothing about the contents. As such, spritesheets can be shared across various items. Typically you should have very few spritesheets to declare. For instance only 4 spritesheets are declared for the entire RPG Maker MV pack (one for the walking sprite, the face sprite, the battler sprite and the damage sprite), and 2 for RPG Maker VX Ace (walk and face).

2. You may want to define portfolios next, as they only depend on spritesheets. Like the spritesheets, portfolios simply define a format, and can be shared across various items. It's a common use case to use a single portfolio that will be common for all the items, and that contain all the spritesheets. However it might happen that you have more convoluted items that require a custom portfolio.
   Note: You may have some items that will not make use of all the spritesheets of the portfolio. Even so, it is OK to assign them with the portfolio. You can see the portfolio as the set of all the spritesheets an item is allowed to use (but not required to). The unusued spritesheets will be automatically stripped from the final item, once exported in GPFI format.

3. Layers also only depend on spritesheets, so we can continue by defining them. Likewise layer objects also define layer properties and can be shared across several items. Layers can be placed into a single spritesheet so you need at least one layer per spritesheet. Typically, you will need to define one layer per spritesheet per "character part". For instance, let's say that you want to export items that extend into two spritesheets: a face sprite and a walk sprite. Each item will contain a character part: be it pants, eyes, hair, shoes, etc... In this case, you may want to define: 2 layers for the pants (one for each spritesheet), 2 layers for the eyes (one for each spritesheet), 2 layers or more for the hair (at least for each spritesheet), etc...
   This is the recommended way as it is common for items that represent the same character part to have the same properties. For instance two different items that represent pants are likely to have layers of similar properties (i.e. same opacity, same blending, etc...). Note that the *position* of a layer is also defined in the layer object. The *position* property can be viewed as the layer height (or z value). The higher it is, the more the layer should be placed at the top, and vice versa. On GCH, layers with a higher *position* are drawn on top of layers with lower *position*. **GCH heavily relies on this property to properly insert items into a character at the right position, without causing superposition issues.** You can make use of this property as well for your own layers and items. This is also a reason why you should define one layer for each character

part (e.g. hair, shoes, ...): usually all the items of a same character part share the same *position*. Finally, note that you might need more than one layer for some "character part". For instance, for the hair, it's common to have a background layer for the hair part that is drawn at the back of the character, and a foreground layer that it drawn on the top of the character. These two layers should have different *position*s, and hence must use two different layer objects.

4. Now that we have defined the foundation of our future items, we may want to think of colorsheets and colorsheet groups. As for the layers, it is advised to define one colorsheet per "character part". The same argument that applied for the layers can be applied for the colorsheets: items of a same character part are usually colorized with the same tints. Colorsheets can be shared across spritesheets, so you don't need to define a colorsheet per spritesheet.
   You may have different generator parts, defined in different items, that should have the same tint (for instance you may have one item that defines a piece of hair, and another that defines a beard. We usually expect the two to have the same tint). To force these two colorsheets to have the same tint, you should create a new colorsheet group, and place then in it.
   Note that at this point, we do not assign layers to the colorsheets. This is done later, when defining the item object.

5. We now have to define the item objects. There must be as many item objects as there GCH items to export (one item object will give one GCH item). Item objects are defined in three steps.
   First, we must define their format: this is done by assigning a portfolio to the item. If the item will support colorsheets, you may also assign them to the item (achieved by gchlib new/set item command). The portfolio (and the colorsheet list) respectively define the set of spritesheets (and colorsheets) which you are allowed to use for the item definition in the next two steps.
   Second, we now have to assign a PNG file to each layer object the item will use (achieved by "gchlib link layer"). gchlib will check that the layer's spritesheet is defined in the item's portfolio. We remind you that you are not required to allocate a PNG image for each layer that are available to the item. Unused layers and spritesheets will be discarded when the item is exported.
   Third, we must define how colorsheets interact with the item (achieved by "gchlib link colorsheet"). You can skip this step if your item does not have any colorsheet. For each colorsheet the item will use, you must declare a list of target layers. These layers must be among the layers we defined in the previous step. This means that the colorsheet will be able to recolor these layers. Also, depending on the colorsheet type (for instance if the colorsheet is based on a palette), you must also declare the path of a PNG file to use as layer tint for each variation allowed by the colorsheet.

6. Optionally, we may want to place our items in a pack. You can skip this step if you do not want to export all the items into a compressed library pack file. This is done by declaring a new pack, and by adding items into it (achieved by the command "gchlib link item"). Now is also a good time to add portfolios, frame mapping processes, animations and character generators (if any) to the pack. These will be installed into GCH when the pack is installed.

7. Finally, that we have completed all the above steps, we are ready to export the items! This step is quite simple, simply run "gchlib make item" or "gchlib make pack". See the commands description in the next chapter.

Now that we have a clearer idea on the steps to perform to export the items, we may go to the next chapter that will explain all these commands.

Chapter 4

gchlib commands overview

4.1 Initialize command

The initialize command creates a gchlib *repository*inside the current working directory (i.e. it creates a ".gchlib" JSON file). The *repository*is where all gchlib intermediary files (e.g. layers, items, etc...) will be referenced.

The *repository*is necessary to run any gchlib command. When gchlib starts, it tries to locate the *repository*. It starts by looking in the current working directory. If it doesn't find it, it will look after it in the parent directory. So on and so forth, gchlib will explore every parent directory until it finds the *repository*, or until it reaches the root of the file system. In the former case, the *repository*of the parent directory will be used. This means that you can run gchlib from any subfolder of your project, assuming that gchlib's *repository*is located at the project root directory. In the latter case, gchlib will fail and will prompt you to run the initialize command.

Command syntax:

```
gchlib init
```

4.2 Add/Remove commands

The "add" and "remove" commands are used to directly add or remove JSON files from the *repository*. See the commands syntax:

```
gchlib add <object-type> <new-reference> <filepath> [-F]
gchlib rm <object-type> <reference> [-c]
```

The first command registers a new file (located at "filepath") of object type "object-type". The file will be referenced as "new-reference" in the *repository*. The command fails if there is already a file of same object type with the same reference in the *repository*. The flag -F removes this protection and overwrites any existing reference from the *repository*.

The second command removes the file of object type "object-type" which is referenced under "reference" in the *repository*. The reference will be removed from the *repository*. The file will also be removed from the disk, unless the -c flag is passed.

In both commands, "object-type" can be any of: "spritesheet", "pack", "layer", "portfolio", "item", "license", "colorsheet", "animation", "generator", "frameMappingProcess", or "colorsheetgroup".

4.3 List command

The list command is able to enumerate all the files referenced in the *repository*. Command syntax:

```
gchlib list <object-type> [-p]
```

This enumerates all the references of type ¡object-type¿. If the flag "p" is set, it also displays the associated file path.

4.4 New/Set commands

The "new" command is used to create a new gchlib object. The "set" command is able to edit a JSON file that is referenced in the *repository*. See the commands syntax:

```
gchlib new <object-type> <new-reference> -p <filepath> [-f] [-F]
gchlib set <object-type> <reference> [-p <filepath>] [-f]
```

The "new" command generates a new JSON file (named "filepath") of type "object-type", which contains the newly created object settings. The file is automatically registered in the *repository*as "new-reference". So you don't need to call "add" afterwards. The command fails if the file "filepath" already exists (unless "f" is set), or if the reference "new-reference" is already in the *repository*(unless "F" is set). The flag "f" will overwrite the file if one already exists. Likewise the flag "F" will overwrite any existing reference from the *repository*. Use "f" and "F" with care.

The "set" command edits the JSON file referenced as "reference". This command can also be used to change the path of the JSON file (with option "p"). Similar to the "new" command, the command fails if such a file already exists, unles the flag "f" is set, in which case gchlib will overwrite it silently.

These commands contain more options to give you control over the object settings. The available options depend on "object-type". See the section below to have the list of all options per object type. In what follows, the options "p", "f" and "F" are omitted for simplicity.

4.4.1 New/Set spritesheet command

Command syntax:

```
gchlib new/set spritesheet <reference> [-n <name>] [-gridw <grid-width> -gridh
<grid-height> -framew <frame-width> -frameh <frame-height> -flv <spritesheet-flavor>]?
[-anim <animation-reference>] [-c <counter>]
```

n Name of the spritesheet. Optional.

gridw Spritesheet grid width. This corresponds to the number of frames that can fit in a row of the spritesheet. Mandatory for the command is "new".

gridh Spritesheet grid height. This corresponds to the number of frames that can fit in a column of the spritesheet. Mandatory for the command is "new".

framew Frame width. This corresponds to the width of a frame (in pixels). Mandatory for the command is "new".

frameh Frame height. This corresponds to the height of a frame (in pixels). Mandatory for the command is "new".

flv Flavor. This corresponds to the flavor of the spritesheet. It is used to classify what kind of animation the spritesheet is for. Mandatory for the command is "new".

anim Spritesheet animation. This corresponds to a reference on an animation. This is the animation that will be played by GCH when the spritesheet is active. Optional.

c Counter. This corresponds to the spritesheet counter. This is used to distinguish otherwise identical spritesheets within a same portfolio. Optional.

4.4.2 New/Set portfolio command

Command syntax:

```
gchlib new/set portfolio <reference> [-n <name>] [-spr <spritesheet-reference>]...
[-reset-sprs]
```

n Name of the portfolio. Optional.

spr Add the spritesheet referenced as *spritesheet-reference* to the portfolio. Can be repeated to add more than one spritesheets.

reset-sprs Clears all the spritesheets from the portfolio. If used with *spr*, the existing spritesheets are removed before processing the *spr* commands.

4.4.3 New/Set layer command

Command syntax:

```
gchlib new/set layer <reference> [-n <name>] [-pos <layer-height>] [-b <blending>]
[-o <opacity>] [-cps <collapse>] [-spr <spritesheet-reference>]
```

n Name of the layer. Optional.

pos Defines the position of the layer. The *position* property can be viewed as the layer height (or z value). The higher it is, the more the layer should be placed at the top, and vice versa. On GCH, layers with a higher *position* are drawn on top of layers with lower *position*. **GCH heavily relies on this property to properly insert items into a character at the right position, without causing superposition issues.**

b Layer blending. Defines how the layer will be drawn on top of the layers of lower position. Among *normal, plus, multiply, screen, overlay, darken, lighten, colordodge, colorburn, hardlight, softlight, difference, exclusion, xor*.

o Layer opacity. From 0 (fully transparent) to 255 (fully opaque).

cps Collapsable. When items are exported, every adjacent and collapsable layer with the same *position* will be merged together. May be *true* or *false*.

spr Associates the layer to the spritesheet referenced as *spritesheet-reference*.

4.4.4 New/Set colorsheet command

A colorsheet object allows recolorization of some parts of an item. It defines the rules of how an item is recolorized. The item parts to recolor are not defined by this command, but via the "link colorsheet" command. The colorsheet object has a *mode* property. This property defines the recolorization technique to use. It must be one of: *auto*, *palette*, *sliders* or *gradient*.

A *palette*-based colorsheet recolorizes by replacing the original layer image by another PNG item. Each tint of a *palette* colorsheet is identified by a color label. The color labels are defined by the *c* option. The amount of color labels indicates the number of tints the colorsheet can support (including the original tint). The colorsheet will need to be supplied with extra PNG files to allow recolorization: one per layer per available tint (but this is achieved via the "link colorsheet" command).

A *sliders*-based colorsheet recolorizes by varying the hue, saturation and lightness of some region of the item. On GCH, the user will be available to control the amount of hue/saturation/lightness to modify via thre sliders. You can control which sliders will be made available to the end user via the *hue*, *sat* and *lum* options.

A *gradient*-based colorsheet using the gradient mapping technique. Gradients are added to the colorsheet by the *g* option. Doing so makes a new tint available to the colorsheet. Basically this works by turning the original item in greyscale, and by replacing black pixels

by the first color of the gradient, white pixels by the last, and various shades of grey by the intermediary colors defined in the gradient.

Declaring a colorsheet with *auto* is just a convenient way to generate a *palette*-based colorsheet. The *auto* colorsheet attempts to automatically generate color variations based on the original item. The amount of available tints is defined by the *count* option. Color variations are generated by varying the hue component of the item. The color labels are computed by averaging the RGB values of the recolorized area.

Command syntax:

```
gchlib new/set colorsheet <reference> [-n <name>] [-m <mode>] [-grp <colorsheet-group>]
[-count <color-count>] [-c <color>] [-reset-cols] [-g <gradient>] [-reset-grds]
[-hue <hue>] [-sat <saturation>] [-lum <lightness>]
```

n Name of the colorsheet. Optional.

m Type of the colorsheet. This defines how the colorsheet will recolorize the layers. Among *auto, palette, sliders, gradient.*

grp Defines the colorsheet group. Optional.

count Only for an *auto* colorsheet. Defines the number of available tints. Optional (defaults to 12).

c Only for a *palette*-based colorsheet. Adds a new tint to the colorsheet. The value must be a color passed in hexadecimal format: "RRGGBB" (e.g. "59D20F" for (R: 89, G: 210, B: 15)), which defines the label color that will be displayed in GCH item library. The first color added this way is special for it is considered as the original tint.

reset-cols Only for a *palette*-based colorsheet. Removes all the tints from the colorsheet.

g Only for a *gradient*-based colorsheet. Adds a new gradient tint to the colorsheet. The parameter must be a gradient. That is it be a list of colors in hexadecimal format, separated by ":". For instance: "000000:FFFFFF" defines a gradient that goes from black to white, and "FF0000:00FF00:0000FF" defines another that goes from red to green and to blue. The list of colors cannot be larger than 256 colors. Unlike for *palette*-based colorsheet, the first gradient does not stand for the original tint, but is indeed considered a new tint like any other.

reset-grds Only for a *gradient*-based colorsheet. Removes all the tints from the colorsheet.

hue Only for a *sliders*-based colorsheet. Value may be *true* or *false*. Activates or deactivates the hue slider in GCH item library explorer.

sat Only for a *sliders*-based colorsheet. Value may be *true* or *false*. Activates or deactivates the saturation slider in GCH item library explorer.

lum Only for a *sliders*-based colorsheet. Value may be *true* or *false*. Activates or deactivates the lightness slider in GCH item library explorer.

4.4.5 New colorsheetgroup command

This command help define colorsheet groups. You may be in the situation where you have different generator parts, defined in different items, but that should have the same tint (for instance you may have one item that defines a piece of hair, and another that defines a beard. We usually expect the two to have the same tint). To make GCH ensures some tint synchronization between these items, you can place the colorsheets responsible for those tints in the same colorsheet group. (For instance, in our previous example, you can first create a new colorsheet group, and the place the colorsheets responsible for the hair color and the one responsible for the beard color inside the newly created group).

Note: You may end up in a situation where you need to have a colorsheet group that contains only one colorsheet. This is not an issue and is perfectly fine. It simply means that GCH will try to enforce the same tint whenever it encounters this colorsheet. (For instance, it is not an issue if in our example, the hair tint and the beard tint are actually handled by the same colorsheet object).

Command syntax:

```
gchlib new colorsheetgroup <reference>
```

There is no "set" version provided for this command as there are no properties to edit.

4.4.6 New/Set item command

Each gchlib item object defines a GCH library item to export in gpfi format. This command only define general item properties. See the "link layer" and "link colorsheet" commands to attach PNG files to the item.

Command syntax:

```
gchlib new/set item <reference> [-n <name>] [-lib <output-path-in-pack>] [-ptf
<portfolio>] [-lic <license>]
```

n Name of the item. Optional.

ptf Reference on the portfolio format to use.

lib Filepath of the item within a pack. You can skip this property if you do not wish to put the item in a pack. The path must contain the subfolders (if any) where the item should be installed from the pack directory, followed by the item filename: e.g. "Male/Cloth/Armor.gpfi". This path is relative to the pack installation directory. This option can be redefined via the "link item" command when the item is addded in a pack.

lic Reference on the license to use for this item. Can be left blank, in which case the license of the pack will be used (if the item is placed in a pack).

4.4.7 New/Set pack command

A pack is a binary file that is used by GCH to install a part of the library. It contains a compressed version of all the items it can install, plus some utilities that it can install as well (frame mapping processes, character generators, spritesheet animations, portfolio format). In this command you may define the pack properties and the utilities it will be installed by the pack.

Note: When a new pack is created via the "new" command, the pack is granted a *unique identifier*. This identifier will appear in clear in the generated JSON file. *It must never be modified for the lifetime of the pack.* GCH relies on this identifier to tell various pack apart. Should the identifier be modified, the pack will be interpreted as a new one by GCH. This may lead to many file conflicts upon the pack installation, and it will mess with GCH's internal files. As such, the golden rule is to **never modify a pack unique identifier**.

Command syntax:

```
gchlib new/set pack <reference> -n <name> -d <output-dir> [-version <version>]
[-lic <license>] [-reset-items] [-fmp <frame-mapping-process-reference] [-reset-fmps]
[-anim <animation-reference>] [-reset-anims] [-ptf <portfolio>] [-reset-ptfs] [-gen
<generator>] [-reset-gens]
```

n Name of the pack. Mandatory.

d Installation directoy of the pack. This path must be relative to the item library installation directory. Mandatory.

version Version of the pack. The value must be a formatted as three numbers, each separated by a dot. (e.g "1.0.0", "0.9.82", etc...).

lic Reference on the license to use for the whole pack. All the items of this pack will use this license unless they overrid it in their definition.

reset-items Removes all the items contained in the pack.

fmp Reference on a frame mapping process to install.

reset-fmps Removes all frame mapping processes from the pack.

ptf Reference on a portfolio format to install.

reset-ptfs Reference all portfolio formats from the pack.

gen Reference on a character generator to install.

reset-gens Reference all character generators from the pack.

anim Reference on a spritesheet animation to install.

reset-anims Reference all spritesheet animations from the pack.

4.4.8 New/Set license command

Command syntax:

```
gchlib new/set license <reference> [-a <author>] [-reset-authors] [-com <policy>]
[-dis <policy>] [-edi <policy>] [-d <description>] [-desc-url <url>] [-lic-url
<url>]
```

a Add a new author to the license. The value is just the author name. Authors must appear by order of importance.

reset-authors Removes all the authors from the license.

com Describes the license terms about commercial use. (is it allowed to use the object of the license for commercial purposes?) Among: *allowed, unallowed, under-conditions, with-link, unspecified.*

dis Describes the license terms about distribution (is it allowed to redistribute the object of the license?). Among: *allowed, unallowed, under-conditions, with-link, unspecified.*

edi Describes the license terms about edition (is it allowed to modify the object of the license?). Among: *allowed, unallowed, under-conditions, with-link, unspecified.*

d Brief description of the license and or the object of the license.

desc-url Internet link to the full description of the license or the license's object.

lic-url Internet link to the full text of the license.

4.5 Link command

The "link" command is used to define the contents of a gchlib item. It is used for instance to populate the item layers and the item spritesheets. See the command syntax:

```
gchlib link <object-type> <item-reference> ...
```

The command will always "link" some gchlib object of type "object-type" to the item referenced as "item-reference". The options offered by the command vary depending on "object-type".

4.5.1 Link layer command

This command is used to associate a PNG file (defined in option *src*, see below) to an item layer (defined in option *lyr*). When the item is exported in GPFI format, this is the file that will be used to fill the layer contents.

The command expects the image to match a specific format. The input image must contain all the frames that are convered by the item, and arranged in a grid layout. The grid

width (amount of frames that fits horizontally) and height (amount of frames that fits vertically) must match the layer's spritesheet grid size.

The command offers a way to disregard some regions of the input image and consider only a subregion of it (see options *subx*, *suby*, *subw*, *subh* to define a subregion with pixel coordinates and *gridx*, *gridy*, *gridw*, *gridh*, *gridcs*, *gridrs* to define it from ratios of the input image). Only the subregion is expected to match the format defined above, Hopefully, thanks to this, you may not have to accomodate your items to gchlib's expected format.

Command syntax:

```
gchlib link layer <item-reference> -lyr <layer-reference> [-n <name>] -src <png-filepath>
[-subx <subregion-x> -suby <subregion-y> -subw <subregion-width> -subh <subregion-height>]
[-gridx <grid-x> -gridy <grid-y> -gridw <grid-width> -gridh <grid-height> [-gridcs
<subregion-width> -gridrs <subregion-height>]]
```

lyr Layer reference (mandatory).

n Layer name (overwrite the one defined on the layer).

src Layer input image (mandatory if new link). Must be set to the PNG file location that will be used to build the layer.

subx X coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

suby Y coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subw Width of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subh Height of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

gridx X coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridy Y coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridw Grid width in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridh Grid height in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridcs Width (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridrs Height (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

4.5.2 Link colorsheet command

This command is used to enable recoloring of an item layer (defined in option *lyr*, see below) via a colorsheet (defined in option *cls*). This command offers many options, whose meaning changes depending on the colorsheet type (i.e. whether the colorsheet is palette-based).

If the colorsheet is palette-based, then the command will save a PNG file in a tint slot defined by the colorsheet. The tint slot is set via the *colid* option. The PNG file to use for the color variation for this layer and tint is defined via the *src* option. The command expects the image to match a specific format. The input image must contain all the frames that are convered by the item, and arranged in a grid layout. The grid width (amount of frames that fits horizontally) and height (amount of frames that fits vertically) must match the layer's spritesheet grid size.

If the colorsheet is not palette-based, then the command simply allows recoloring for the specified layer. The command allows you to limit the recolorization to only a part of the original layer image. This area where recoloring is authorized can be defined with pixel precision. This is achieved by passing an extra PNG file to the command, via the *src* option. This input image will be used as a mask. You will also need to define an inclusive or exclusive color (options *in-msk* or *ex-msk*). If the inclusive color is set, then the recoloring area will be defined by all the pixels whose color matches *in-msk*. On the contrary, if the exclusive color is set, then the recoloring area will be defined by all the pixels whose color does not match *ex-msk*. Note that if you rely on a mask, you should make sure that the mask image size matches the original layer input image size.

The command offers a way to disregard some regions of the input image (be it a mask or a whole tint) and consider only a subregion of it (see options *subx*, *suby*, *subw*, *subh* to define a subregion with pixel coordinates and *gridx*, *gridy*, *gridw*, *gridh*, *gridcs*, *gridrs* to define it from ratios of the input image). Only the subregion is expected to match the format defined above, Hopefully, thanks to this, you may not have to accomodate your items to gchlib's expected format.

Command syntax:

```
gchlib link colorsheet <item-reference> -cls <colorsheet-reference> -lyr <layer-reference>
[-colid <tint-id>] [-in-msk <inclusion-color>] [-ex-msk <exclusion-color>] [-n
<name>] [-src <png-filepath>] [-subx <subregion-x> -suby <subregion-y> -subw <subregion-wi
-subh <subregion-height>] [-gridx <grid-x> -gridy <grid-y> -gridw <grid-width>
-gridh <grid-height> [-gridcs <subregion-width> -gridrs <subregion-height>]]
```

cls Colorsheet reference (mandatory). Link the referred colorsheet to the item.

lyr Layer reference (mandatory).

colid Color variation number (starts at 1, 0 being the original image. Mandatory if the colorsheet type is "palette").

ex-msk Define the mask exclusion color. The color variation will apply only on the region defined by the mask pixels whose color is NOT "ex-msk" (mandatory if option "src" is set and for a colorsheet of type "sliders", "gradient" or "auto").

in-msk Define the mask inclusion color. The color variation will apply only on the region defined by the mask pixels whose color is "in-msk" (mandatory if option "src" is set and for a colorsheet of type "sliders", "gradient" or "auto").

n Colorsheet name (overwrite the one defined on the colorsheet).

src Colorsheet input image file location. For a colorsheet of type "palette", the image will be used to build a new tint (numbered by "colid") for this layer. For a colorsheet of type "sliders", "auto" or "gradient", this parameter is optional and is used to define a mask. The mask s then used to determine on which regions of the layer the color variation should apply.

subx X coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

suby Y coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subw Width of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subh Height of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

gridx X coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridy Y coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridw Grid width in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridh Grid height in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridcs Width (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridrs Height (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

4.5.3 Link item command

This command is used to define various item meta data. This commands contrasts against the "new/set item" commands for they only define format-related item properties. This command enables you to:

- Add the item to a pack. See *pck* and *lib* options.
- Define the path where the item will be exported if exported via the "make item" command. See the *out* option.
- Define a snapshot for the item. The snapshot will be displayed as a file icon in GCH item library explorer. See the *snapshot* option, and the various *sub**, *grid** commands.

Command syntax:

```
gchlib link item <item-reference> [-out <export-filepath>] [-pck <pack-reference>]
[-lib <item-path-in-library>] [-snapshot <png-filepath>] [-subx <subregion-x> -suby
<subregion-y> -subw <subregion-width> -subh <subregion-height>] [-gridx <grid-x>
-gridy <grid-y> -gridw <grid-width> -gridh <grid-height> [-gridcs <subregion-width>
-gridrs <subregion-height>]]
```

out Output file location. This corresponds to the path where the item will be exported if exported via the "make item" command.

pck Pack reference. Add the item into the referenced pack.

lib Item filename in the item library. It corresponds to the location where the item will be installed within the item library, if installed from the pack referenced as "pck". The filename must be relative to the pack installation folder. If this option is set, the filename defined on the item level is ignored and this one is used instead.

snapshot PNG file location that will be used as an item preview.

subx X coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

suby Y coordinate of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subw Width of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

subh Height of the pixel subregion of the input image. If defined, only the subregion will be exported into the layer.

gridx X coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridy Y coordinate of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridw Grid width in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridh Grid height in which the input image lies. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridcs Width (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

gridrs Height (in frames) of the grid-based subregion of the input image. The grid virtually divides the input image into frames (via "gridw" and "gridh"). The selected frames (via "gridx", "gridy", "gridcs", "gridrs") defines a subregion. If defined, only the selected frames will be exported into the layer.

4.6 Make command

The *make command* is used to export the items into GPFI files or a GCH library pack.

4.6.1 Make item command

The *make item* command can export a list of pre-configured items into GPFI files. The gpfi file (output file) will be exported at the location specified in the item's "output" property. If the nothing is specified, then the output file will be exported along with the JSON definition file of the item, with the same filename, except for the file suffix which will be set to "gpfi" (e.g. the item JSON file "Elf/Body.json" will be exported into "Elf/Body.gpfi"). The generated items can then simply be placed inside the item library folder to be available in GCH.

See command syntax:

```
gchlib make item [-pck <pack>] [-item <item>]...
```

pck This will create all the items that are contained in the pack referenced as "pck".

item May be used several times to export several items. Export the items given as parameters.

4.6.2 Make pack command

The *make pack* command is used to export a pre-configured gchlib pack.

See command syntax:

```
gchlib make pack <pack-ref> <pack-output-path>
```

The command will export the pack referenced as "pack-ref" to the file "pack-output-path". The ".gchpack" file suffix will be automatically added to the output filename if missing. The generated pack can be placed inside the "itempacks" folder from GCH's installation directory. Upon launch, GCH will ask the user if they want to install the pack.

Note: Each pack contains a version number. GCH will rely on it to know whether a pack has been updated. Hence if you made any change in your pack, you will need to upgrade its session number for GCH to detectg a change.

4.7 Dump command

This command is supplied for debugging purposes. It allows to read the binary that gchlib generated (the GPFI items and the GCHPACK files).

4.7.1 Dump item command

This command displays information about a previously generated gpfi file. It can display information about its spritesheets, layers and colorsheets (in that order). You can use this

command to make sure that an item was properly exported. See command syntax:

```
gchlib dump item <gpfi-file-location> [-spr] [-lyr] [-cls]
```

spr Display spritesheet information (enabled if no option is set). Information is displayed as table where is cell is separated by a tabulation (
t). The table columns are: id, name, grid size, frame size, flavor, counter, animation. The id is a unique identifier assigned to each spritesheet by gchlib.

lyr Display layer information (enabled if no option is set). Information is displayed as table where is cell is separated by a tabulation (
t). The table columns are: owner spritesheet id:layer id, name, position, opacity, blending, collapsable. "owner spritesheet id" indicates the spritesheet which the layer belongs to. To be used with the *spr* option if you want to know which spritesheet corresponds to this id. The layer id is a unique identifier of a layer within a spriteshet and is assigned by gchlib.

cls Display colorsheet information (enabled if no option is set). Information is displayed as table where is cell is separated by a tabulation (
t). The table columns are: name, colorsheet group, type, comma-separated list of target layers (formatted as "spritesheet-id:layer-id").

4.7.2 Dump pack command

This command displays information about a previously generated GCH library pack (a gchpack file). The command simply displays all the files contained in the pack, followed by their size. See command syntax:

```
gchlib dump pack <gchpack-file-location>
```

4.8 Execute command

This command allows you *run a gchlib script*. A *gchlib script* is a text file where each line contains a gchlib command. Lines that start with a "#" are ignored and can be used to comment the script file. See here an example of a gchlib script:

```
   # File "MyItem.desc" # Create spritesheet
new spritesheet walk -p spr.walk.json -flv walk -gridw 3 -gridh 4 -framew 48 -frameh
48 -f -F
# Create portfolio
new portfolio mypack_portfolio -p ptf.json -spr walk -f -F
# Create layer
new layer body -p lyr.body.json -spr walk -f -F
```

```
# Create item
new item item_body -p item.body.json -ptf mypack_portfolio
# Add PNG file to item
link layer item_body -lyr body -src "Male/Body.png"
# Export the item
make item -item item_body
```

This script can be run via: "gchlib exec MyItem.desc". Running this script is equivalent to running each command it contains individually. This command provides some advantages:

- Performance consideration: gchlib will run faster if the commands are executed from a script. This is for 3 reasons:
  1. Upon start, gchlib has to read the repository. If all the commands you want to play are in a script, then gchlib will have to read the repository only once compared to once per command.
  2. Likewise, when quitting, gchlib refreshes all the files that were touched by a command. This operation is performed concurrently, dispatched on all the available cores of the CPU. It is then faster to refresh a large bunch of files at the same time (via the execute command) instead of refreshing the same amount of files but a few at a time (if each command is played separately).
  3. gchlib heavily relies on references. It's common to have a file rely on another one, and in such case we say that the first file "references" the other. Hence it is fairly common to have gchlib read several files for simple commands, because it often needs to read the files that are referenced. To speed things up, gchlib remembers the files it has already read and will not read them again until it terminates. Thus running all commands via a script will make gchlib remember all the files it read for the whole script and as such it will spend less time re-reading files.
- Convenience: Most gchlib commands are self-explanatory and have an easy-to-read syntax. They really define and describe how the items are created. We find it very convenient to have all the commands in a descriptive file that allows comment (a gchlib script).
- Organization: A gchlib script is free to invoke another script. Hence you can divide your commands into several scripts and have a master script that will call each of them.

Chapter 5

Appendix: grammar of json objects

5.1 Json grammar of the Repository

```
Json object {
     "layers":  Json array [
          Json array of size 2 [
               string (reference on layers),
               string (filepath)
          ]
     ],
     "spritesheets":  Json array [
          Json array of size 2 [
               string (reference on spritesheets),
               string (filepath)
          ]
     ],
     "portfolios":  Json array [
          Json array of size 2 [
               string (reference on portfolios),
               string (filepath)
          ]
     ],
     "colorsheets":  Json array [
          Json array of size 2 [
               string (reference on colorsheets),
               string (filepath)
          ]
     ],
     "items":  Json array [
          Json array of size 2 [
               string (reference on items),
               string (filepath)
          ]
     ],
```

```
     "animations":  Json array [
          Json array of size 2 [
               string (reference on animations),
               string (filepath)
          ]
     ],
     "generators":  Json array [
          Json array of size 2 [
               string (reference on generators),
               string (filepath)
          ]
     ],
     "licenses":  Json array [
          Json array of size 2 [
               string (reference on licenses),
               string (filepath)
          ]
     ],
     "frameMappingProcesses":  Json array [
          Json array of size 2 [
               string (reference on frameMappingProcesses),
               string (filepath)
          ]
     ],
     "packs":  Json array [
          Json array of size 2 [
               string (reference on packs),
               string (filepath)
          ]
     ],
     "colorsheetgroups":  Json array [
          Json array of size 2 [
               string (reference on colorsheetgroups),
               string (filepath)
          ]
     ]
}
```

5.2 Json grammar of object type ”layer”

```
Json object {
     "name":  string,
     "opacity":  integer with value between 0 and 255,
     "position":  integer with value between -50 and 50,
     "blending":  any of:  <normal, plus, multiply, screen, overlay, darken, lighten,
```

```
colordodge, colorburn, hardlight, softlight, difference, exclusion, xor>,
     "spritesheet":  string (reference on spritesheets),
     "collapsable":  boolean
}
```

5.3 Json grammar of object type "spritesheet"

```
Json object {
     "grid":  Json array of size 2 [
          integer (width in frames) with value between 1 and 1024,
          integer (height in frames) with value between 1 and 1024
     ],
     "frame":  Json array of size 2 [
          integer (width in pixels) with value between 8 and 4096,
          integer (height in pixels) with value between 8 and 4096
     ],
     "flavor":  any of:  <walk, run, jump, crouched, face, fight, spellcraft, item-use,
crafting, humor, wounded, dead, other>,
     "counter":  non-negative integer with maximum:  255,
     "name":  string,
     "animation":  string (reference on animations)
}
```

5.4 Json grammar of object type "portfolio"

```
Json object {
     "name":  string,
     "spritesheets":  Json array [
          string (reference on spritesheets)
     ]
}
```

5.5 Json grammar of object type "colorsheet"

```
Json object {
     "name":  string,
     "group":  string (reference on colorsheetgroups),
     "mode":  any of:  <auto, palette, sliders, gradient>,
     "hue":  boolean,
     "sat":  boolean,
     "lum":  boolean,
     "count":  integer with value between 1 and 32,
     "colors":  Json array [
```

```
          Json array (RGBA color) [
               integer (red component), with value between 0 and 255,
               integer (green component), with value between 0 and 255,
               integer (blue component), with value between 0 and 255,
               integer (alpha component, optional), with value between 0 and 255.
255 if missing
          ]
     ] with size between 0 and 32,
     "gradients":  Json array [
          Json array [
               Json array (RGB color) [
                    integer (red component), with value between 0 and 255,
                    integer (green component), with value between 0 and 255,
                    integer (blue component), with value between 0 and 255
               ]
          ] with size between 1 and 256
     ] with size between 0 and 255
}
```

5.6 Json grammar of object type ”item”

```
Json object {
     "name":  string,
     "lib-path":  string,
     "output":  (nil) or string,
     "portfolio":  string (reference on portfolios),
     "license":  string (reference on licenses),
     "shapshot":  (nil) or Json object {
          "src":  integer (column span),
          "px-subrect":  (nil) or Json array of size 4 [
               integer (X coordinate),
               integer (Y coordinate),
               integer (width),
               integer (height)
          ],
          "gd-subrect":  (nil) or Json array of size 6 [
               integer (X coordinate) with minimum:  0,
               integer (Y coordinate) with minimum:  0,
               integer (column span) with minimum:  1,
               integer (row span) with minimum:  1,
               integer (width) with minimum:  1,
               integer (height) with minimum:  1
          ]
     },
     "layers":  Json array [
```

```
          Json array of size 2 [
               string (reference on layers),
               Json object {
                    "src":  integer (column span),
                    "px-subrect":  (nil) or Json array of size 4 [
                         integer (X coordinate),
                         integer (Y coordinate),
                         integer (width),
                         integer (height)
                    ],
                    "gd-subrect":  (nil) or Json array of size 6 [
                         integer (X coordinate) with minimum:  0,
                         integer (Y coordinate) with minimum:  0,
                         integer (column span) with minimum:  1,
                         integer (row span) with minimum:  1,
                         integer (width) with minimum:  1,
                         integer (height) with minimum:  1
                    ],
                    "name":  (nil) or string
               }
          ]
     ],
     "dyes":  Json array [
          Json array of size 2 [
               string (reference on colorsheets),
               Json object {
                    "name":  (nil) or string,
                    "targets":  (nil) or Json array [
                         Json array of size 2 [
                              string (reference on layers),
                              (nil) or Json object {
                                   "src":  integer (column span),
                                   "px-subrect":  (nil) or Json array of size 4
[
                                        integer (X coordinate),
                                        integer (Y coordinate),
                                        integer (width),
                                        integer (height)
                                   ],
                                   "gd-subrect":  (nil) or Json array of size 6
[
                                        integer (X coordinate) with minimum:  0,
                                        integer (Y coordinate) with minimum:  0,
                                        integer (column span) with minimum:  1,
                                        integer (row span) with minimum:  1,
                                        integer (width) with minimum:  1,
                                        integer (height) with minimum:  1
```

```
                                    ],
                                    "color":  Json array (RGBA color) [
                                         integer (red component), with value between
0 and 255,
                                         integer (green component), with value between
0 and 255,
                                         integer (blue component), with value between
0 and 255,
                                         integer (alpha component, optional), with
value between 0 and 255.  255 if missing
                                    ],
                                    "inclusive":  boolean
                               }
                          ]
                     ],
                     "palette-targets":  (nil) or Json array [
                          Json array [
                               Json array of size 2 [
                                    string (reference on layers),
                                    Json object {
                                         "src":  integer (column span),
                                         "px-subrect":  (nil) or Json array of size
4 [
                                              integer (X coordinate),
                                              integer (Y coordinate),
                                              integer (width),
                                              integer (height)
                                         ],
                                         "gd-subrect":  (nil) or Json array of size
6 [
                                              integer (X coordinate) with minimum:
0,
                                              integer (Y coordinate) with minimum:
0,
                                              integer (column span) with minimum:
1,
                                              integer (row span) with minimum:  1,
                                              integer (width) with minimum:  1,
                                              integer (height) with minimum:  1
                                         ]
                                    }
                               ]
                          ]
                     ]
                }
           ]
      ]
```

```
}
```

5.7 Json grammar of object type ”animation”

```
Json object {
     "name":  string with not null size,
     "grid":  Json array of size 2 [
          integer (width in frames) with value between 1 and 1024,
          integer (height in frames) with value between 1 and 1024
     ],
     "animations":  Json array [
          Json object {
               "name":  string with not null size,
               "frames":  Json array [
                    Json object {
                         "x":  non-negative integer (x coordinate) with maximum:
4095,
                         "y":  non-negative integer (y coordinate) with maximum:
4095,
                         "duration":  non-negative integer (duration in ms) with
value between 50 and 60000
                    }
               ]
          }
     ]
}
```

5.8 Json grammar of object type ”generator”

```
Json object {
     "name":  string,
     "portfolio":  Json array [
          Json object {
               "grid":  Json array of size 2 [
                    integer (width in frames) with value between 1 and 1024,
                    integer (height in frames) with value between 1 and 1024
               ],
               "frame":  Json array of size 2 [
                    integer (width in pixels) with value between 8 and 4096,
                    integer (height in pixels) with value between 8 and 4096
               ],
               "flavor":  any of:  <walk, run, jump, crouched, face, fight, spellcraft,
item-use, crafting, humor, wounded, dead, other>,
               "counter":  non-negative integer with maximum:  255,
```

```
            "name":  string
        }
    ],
    "parameters":  Json array [
        Json object {
            "name":  string,
            "ref":  bytefield,
            "values":  Json array [
                Json object {
                    "name":  string,
                    "ref":  bytefield,
                    "weight":  integer with value between 0 and 100
                }
            ]
        }
    ],
    "layers":  Json array [
        Json object {
            "name":  string,
            "layers":  Json array [
                Json object {
                    "src":  string,
                    "weight":  integer
                }
            ] with maximum size 65536,
            "color":  any of:  <none, randomize>,
            "subfolders":  boolean,
            "spawn":  Json object {
                "rate":  integer with value between 0 and 100,
                "enable-conditions":  boolean,
                "conditions":  Json array of format "LogicalRules" [
                    1st item:  any of:  <all, any, equal, different>,
                    If first item is "all" or "any", any remaining items:
Json array of format "LogicalRules",
                    If first item is "equal" or "different", second item is:
string (parameter id),
                    If first item is "equal" or "different", third item is
the last one and is:  string (parameter value id)
                ]
            }
        }
    ]
}
```

5.9 Json grammar of object type "license"

```
Json object {
     "editing":  any of:  <allowed, unallowed, under-conditions, with-link, unspecified>,
     "distribute":  any of:  <allowed, unallowed, under-conditions, with-link,
unspecified>,
     "commercial-use":  any of:  <allowed, unallowed, under-conditions, with-link,
unspecified>,
     "authors":  Json array [
          string
     ],
     "description":  string,
     "description-url":  string (formatted as url),
     "license-url":  string (formatted as url)
}
```

5.10 Json grammar of object type "frameMappingProcess"

```
Json object {
     "name":  string,
     "src-grid":  Json array of size 2 [
          integer (width in frames) with value between 1 and 1024,
          integer (height in frames) with value between 1 and 1024
     ],
     "dst-grid":  Json array of size 2 [
          integer (width in frames) with value between 1 and 1024,
          integer (height in frames) with value between 1 and 1024
     ],
     "transform":  any of:  <none, frame, rescale>,
     "dst-frame":  Json array of size 2 [
          integer (width in pixels) with value between 8 and 4096,
          integer (height in pixels) with value between 8 and 4096
     ],
     "mapping":  Json array [
          Json array of size 2 [
               Json array of size 2 [
                    integer (X coordinate),
                    integer (Y coordinate)
               ],
               Json array of size 2 [
                    integer (X coordinate),
                    integer (Y coordinate)
               ]
          ]
     ] with maximum size 1048576
```

```
}
```

5.11 Json grammar of object type ”pack”

```
Json object {
     "name":  string,
     "dir":  string,
     "reserved-id":  bytefield,
     "version":  string (formatted as a version number, i.e.  a dot-separated list
of digits),
     "items":  Json array [
          Json array of size 2 [
               string (reference on items),
               Json object {
                    "lib-path":  string
               }
          ]
     ],
     "license":  string (reference on licenses),
     "frameMappingProcesses":  Json array [
          string (reference on frameMappingProcesses)
     ],
     "animations":  Json array [
          string (reference on animations)
     ],
     "portfolios":  Json array [
          string (reference on portfolios)
     ],
     "generators":  Json array [
          string (reference on generators)
     ]
}
```

5.12 Json grammar of object type ”colorsheetgroup”

```
Json object {
     "reserved-id":  string
}
```