# Sonic 3 A.I.R. – Angel Island Revisited

# Modding Documentation

## Contents

# General Information

## Where to put mods?

If you want to install a mod, start by opening the **Sonic3AIR saved data folder**, either:

1. Start the game, **right-click** into the game window and then select the button "Open saved data folder"
2. or do it manually by opening the Windows Explorer and entering "%appdata%/Sonic3AIR" (without the quotes) into the address line

Mods have to be placed into the **"mods"** subfolder in there.

## What parts of the game can be modded?

At the moment, modding allows for changes to:

1. Game music
2. Level contents
3. Character and HUD sprites
4. Character palettes
5. Overwriting parts of the ROM (raw data modding)
6. Game functionality (script modding)

See further below for instruction how to make these kinds of mods.

## Is certain prior knowledge needed for modding?

It depends on the kind of modding you want to get into.
For instance, audio modding requires some basic experience with audio editing software, and for level modding you need to know the SonLVL editor.
On the other hand, raw data and script modding won't be feasible with a good knowledge of Sonic ROM hacking or general programming skills.

Aside from that, a good general technical understanding is definitely helpful.
This includes the ability to **edit JSON text files**, which is needed for all types of mods. I'm not going to explain the details here, and small mistakes like missing commas can result in the game just stopping with an error message (or worse) instead of loading your mod.

## How to setup the basic structure of a mod

**To create a new mod:**

- Open "%appdata%/Sonic3AIR/mods" in Windows Explorer and create **a new folder** there. As folder name, use whatever you want your mod to be named.
- Create a JSON text file named **"mod.json"** and fill it with metadata like the mod name and author, just like this:

```
{
	"Metadata":
	{
		"Name": "Heavy & Bomb",
		"UniqueID": "euka-heavy-bomb-characters",
		"Author": "Euka",
		"Description": "Adds Heavy & Bomb from Chaotix as playable characters.",
		"URL": "https://yourhomepage.com",
		"ModVersion": "1.0.0.0",
		"GameVersion": "23.04.02.0"
	}
}
```

- Alternatively, copy one of the **sample mods** from the "sample-mods" folder.

**Some details on the mod.json contents**

Inside the "Metadata" section, add the following entries as shown above:

- "Name": The displayed name of your mod, shown in menus.
- "UniqueID": An ID string that is unique to your mod. This is sort of an internal / technical identifier for your mod that can be used to identify it unambiguously and across multiple mod updates. You can choose this ID freely, just make sure you don't use the same unique ID as another existing mod. A good idea might be to include your nickname.
- "Author" (optional): Your nickname.
- "Description" (optional): Some brief description of what your mod does.
- "URL" (optional): Can be the URL of your homepage or social media profile.
- "ModVersion": Give your mod a version number. It's a good idea to update this when you release a new version of your mod so people know which version they are using.
- "GameVersion": The minimum Sonic 3 A.I.R. game version that is required to run your mod.

**More meta data**

It's also recommended to **add an icon** to your mod. To do so, add one or more PNG files using the following file names, and place them directly next to the "mod.json" file.

- "icon.png"
  - High resolution icon used by the Mod Manager, and as fallback in the game itself if none of the other icons below are present.
  - Can be any image size between 64x64 and 256x256 pixels.
- "icon-64px.png"
  - Large icon shown in the details view of the game's Mods menu.
  - Should be exactly 64x64 pixels.
- "icon-16px.png"
  - Small icon shown in the game's Mods menu.
  - Should be exactly 16x16 pixels.

Make sure to export all PNGs as either RGB or RGBA with 8 bits per channel and no interlacing.

## Additional sections in your mod.json file

In addition to the "Metadata" section as shown above, there's also some optional sections you might want to add as well to your "mod.json":

- **"Settings"** allows you to add your own entries to the game's Options menu, in the Mods tab. For details on this, see the chapter Using Custom Mod Settings.
- **"UsesFeatures"** enabled certain engine features if any active mods lists the feature in this section.
  - **"Controls_LR"**: By default, S3AIR supports the following buttons for mods to use: Left, right, up, down, A, B, X, Y, Start, Back. If you add "Controls_LR" and set it to true, this is extended by L and R shoulder buttons. Those will appear in the controls setup menu and in the mobile versions' touch input overlay.

    If you want to use these extra buttons in your mod, add this line into your mod.json: `"UsesFeatures": { "Controls_LR": true }`
- **"OtherMods"** is useful if your mod either has some dependency on another mod. Or if you need to define the correct order of mods in order to make your mod work with certain other mods. See below for the details.

### Interactions with other mods

If your mods requires another mod to work correctly, add a section like this:

```
"OtherMods":
{
	"euka-base-script-mod":
	{
		"DisplayName": "Euka's Base Script Mod",
		"IsRequired": "1",
		"Priority": "Lower"
	},
	"euka-sample-sprite-mod":
	{
		"DisplayName": "Sample Sprite Mod",
		"IsRequired": "0",
		"Priority": "Higher"
	}
}
```

This would tell the game about your mod's dependencies and interactions with two other mods. Namely one named "Euka's Base Script Mod", which is required by your mod. And the "Sample Sprite Mod", which is not required, but should be using a higher mod priority if activated together with your own mod.

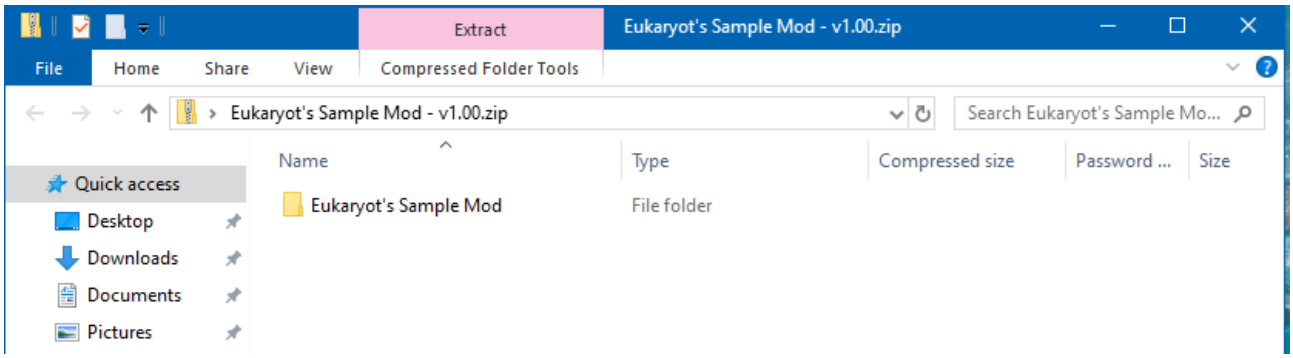
The format in detail:

- You can add information about one or multiple mods in the `"OtherMods"` section. All of this is completely optional, but might be useful for players, as the game will show warnings about missing requirements and wrong order of mods in the Mods menu.
- The key (`"euka-base-script-mod"` or `"euka-sample-sprite-mod"` in the example above) for each entry is that mod's unique ID. It can be found inside the respective mod.json, under "Metadata" -> "UniqueID". If the mod doesn't have a unique ID (yet), you can also use its display name instead.

- `"DisplayName"` is only used so that the game can show a more readable name for the mod to the player, in case that mod is not installed yet.
- `"IsRequired"` can be either:
  - `"1"` if your mod won't work at all without that other mod.
  - `"0"` if the other mod is not required, but you still want to give the player a hint about the right order of mods, in case the player has both of them activated.
- `"Priority"` defines where the player needs to place that mod relative to your own mod. It can be either:
  - `"Lower"` if the the player needs to place the other mod at a lower priority than your own mod, i.e. anywhere below it in the Mods menu.
  - `"Higher"` if the the player needs to place the other mod at a higher priority than your own mod, i.e. anywhere above it in the Mods menu.

## Prepare your mod for distribution

When your mod is finished, it's strongly recommended packing the **whole mod folder** as a ZIP file. So when you open the ZIP file after packing, the first thing you see is a single folder named like the mod itself.
Just like that:



# Testing & Debugging

## Config settings

Especially when making level mods, you certainly want to be able to quickly start into the game and test your changes. There are some hidden options supported in the **"config.json"** that is located in the game installation's base folder. To enable them, open this file with a text editor and add one or more of the following lines inside the `"DevMode"` section:

`"Enabled": "1",`

Enables **Dev Mode**. This includes the Debug Mode as known from original Sonic 3 & Knuckles (it can be unlocked by completing achievements as well, but Dev Mode will enable it in any case), and it enables certain debugging keys and features as described in the Oxygen handbook PDF.

`"StartPhase": "2",`

Skips more or less steps of the game, like disclaimer and title screen.
As parameter use one of the following:

- `"0"` -> Start the game as usual, beginning with the disclaimer
- `"1"` -> Start with the SEGA screen, skipping only the disclaimer
- `"2"` -> Start with the main menu, skipping everything before
- `"3"` -> Start right into the game, which makes most sense when combined with loading a save state

This option has no effect if `"LoadLevel"` is used.

`"LoadLevel": "0x0701",`

Skip all menus and directly load a level. The parameter uses hexadecimal notation of the zone and act in the form `"0xZZAA"`.

- For ZZ, use the zone's number in internal a.k.a. "Prototype" zone ordering:
  00 is Angel Island, …, 03 is Carnival Night, 04 is Flying Battery, 05 is Ice Cap, …, 09 is Lava Reef.
- For AA, only 00 or 01 is valid, for Act 1 and 2, respectively.

The example above would load Mushroom Hill Act 2. Note that this won't work with invalid or incomplete zone/act combinations.

`"UseCharacters": "3",`

To be used with `"LoadLevel"`. Specify which character(s) you want to start the level with:

- `"0"` -> Sonic & Tails
- `"1"` -> Sonic
- `"2"` -> Tails
- `"3"` -> Knuckles
- `"4"` -> Knuckles & Tails (if unlocked)

## Debugging keys

The following keys require the Dev Mode to be activated first, see above.

- **F10**: Reload modded resources, including sprites and palettes
  *Note: This does not necessarily have an immediate effect. For instance, character palettes get loaded on level initialization, so might have to do a restart from the Pause Menu to see palette changes.*
- **Alt + M**: Toggle palette debug output

# Music Modding

## Requirements

You can change the game's music and jingles.
To do so, two things are needed:

- One or more music files in **Ogg Vorbis** format.
- A **JSON file** telling the game which music to replace using which file.

Music file requirements:

- Must use Ogg Vorbis Format (**.ogg**) – you can use e.g. Audacity for Windows, a freeware audio processing application, to convert MP3 or similar to this format.
- It's recommended to use Stereo and a **48000 Hz** sampling rate.
- Choose a filename as you like, just avoid non-ASCII characters.

## How to mod

First of all, **setup your mod** as described under "How to setup the basic structure of a mod" in the General "Information section" above.
To start with your music mod, now go through the following steps:

1. Create a subfolder **"audio"** inside your mod.
2. Place all music files you want to use in there.
3. Copy the template file **"audio_replacements.json"** inside "modding/audio" next to this document into your mod's "audio" folder.
4. Open the JSON file with a text editor and fill in the right content.
   - **Remove** all lines referring to music tracks you *don't* want to replace; the comment should tell you which lines are the right ones.
   - Replace all ... with the **names of your music files**, and leave the rest as it is; a line should look like that afterwards:
     `"0F": { "File": "mhz1.ogg", "Type": "Music" },`
5. When you're done, start the game to test your mod.

Some details on how the JSON lines are built:

- The **key** (e.g. `"0F"`) is the sound ID that line refers to. You can get the right one from the game option's Sound Test, or use the sample "audio_replacements.json" as a reference.
- The **file** (e.g. `"mhz1.ogg"`) is the name of the Ogg Vorbis file to be used as modded music track. Make sure to place all Ogg Vorbis files in the same folder as the "audio_replacements.json".
- For the other properties like `"Type": "Music"` leave them exactly as they are in the template.
- You might need to add more properties, especially for `"LoopStart"`, see below.

## Loop Start

To add proper looping of music tracks:

- You can optionally define a position where a music tracks starts looping again just after reaching the end. Just add the `"LoopStart"` tag like in the following example:
  `"0F": { "File": "mhz1.ogg", "Type": "Music", "LoopStart": "432090" },`
- The "LoopStart" parameter value is the **number of samples** (i.e. 1/48000 of a second) from the audio file's start. (Audacity allows you to display the selected position in frames, which makes getting the right value easier.)

## Fast Music Tracks

In certain cases, the game speeds up the level or boss music tracks – for example when opening a speed shoes monitor. For these cases, a separate faster music track is needed.

What you need to know:

- A fast music track has to be **exactly 25% faster** than the normal music, with a length of 0.8 (= 1 / 1.25) the normal length and 0.8 times the loop start position. This is needed so that the game can change between normal and fast tracks at any time.
- An easy way to create fast music tracks is by doing an automatic conversion. Audacity can do that with "Effect -> Change Tempo" and a "change in %" value of 25. (This is not optimal quality-wise, but better than nothing.)

How to add a fast music track to the game:

- Add a separate line into "audio_replacements.json" for **each fast music** track.
- As **key**, use the same as for the normal track, but with `_fast` added at the end:
  `"0F_fast": { "File": "mhz1_fast.ogg", "Type": "Music", "LoopStart": "345672" },`

## Details on audio_replacements.json

If you're interested in diving deeper into audio modding, here's the details on how the JSON lines in audio_replacements.json are built.

The key can be an arbitrary string, so you can choose whatever you like and reference that key string in a script like inside the "Standalone.playAudio" function.
Be aware though that some key strings have a special meaning:

- A two-character string that is a hexadecimal value (like `"6A"`) is automatically associated with one of S3&K's original sound IDs. Using these, you will in almost all cases overwrite an existing music track or sound effect.
- If the string ends with `_fast`, it's regarded as a fast music version – see above.

There's the following available properties for each key:

- "File"
  - The file name of an Ogg Vorbis file to play.
  - Or alternatively, a SMPS file for emulated playback. But this also requires setting a "Source", see below.
  - Leave this property out completely for emulated playback from ROM.
- "Source"
  - Set this only for emulated playback, not for playback from an Ogg Vorbis file.
  - If not using an Ogg Vorbis file as source, this can be one of the sub-types "Emulation", "EmulationDirect" or "EmulationContinuous".

    - The engine will then either:
        - play back original SMPS data from the ROM
        - or from a SMPS file, if one was set with the "File" property
    - Differences between the sub-types:
        - "Emulation" plays and caches the sound data. Used when playback of a track is always the same, like it‘s the case for many sound effects.
        - "EmulationDirect‟ plays the sound data without any caching. This is the right choice when playback can differ depending on what happens in the game – e.g. level music tracks need to support being sped up in between while having speed shoes.
        - "EmulationContinuous" is meant specifically for certain continuous sound effects. This way, the sound does not get restarted by a "Audio.playAudio" script call, but just gets a signal to continue playing.
- "Type"
    - This is one of "Music", "Jingle" or "Sound".
    - It determines the basic behavior of the sound, e.g. whether it loops and if it counts as music or sound effect for volume control.
- "Channel"
    - Selects a playback channel to use. For each channel, there can be only one track playing at a time, so starting a new track will stop the old one, if one plays already.
    - Leave this out to use no channel at all, i.e. unrestricted playback independent of all other playing tracks.
    - Music and jingles automatically use channel 0, so this onyl has an effect on "Sound" type.
- "LoopStart"
    - Start position in samples for the looping part of the track. Only relevant for the "Music" type.
- "Volume"
    - Sets a relative playback volume as a float. This is "1.0" by default, i.e. normal volume.
- "Address"
    - For emulated playback from ROM only: Can be used to override the ROM address of a track.
- "ContentOffset"
    - For SMPS file playback only: Actual start of SMPS data for the track inside the given SMPS file.
    - This is meant for cases when the track doesn't start at the very beginning of the file already - e.g. because there's more than one track in the same file.
- "EmulatedID"
    - For emulated playback (either ROM or SMPS file): Set the original sound ID associated with the track.
    - This is needed because the S3&K sound driver makes some hard-coded differentiation inside depending on this ID.
    - It‘s a value between "01" and "FF".

# Level Modding

*Please note: Starting with version 19.12.15.0, the folder structure for level mods has changed, and support for the old format will be removed with a future version of S3AIR.*
*The old format used entries in the "mod.json", whereas the new format uses the raw data mechanism to reference binary files. See details below and in the "Raw Data Modding" section.*

## Preparations

Before you begin, **setup the SonLVL editor** and the Sonic 3 & Knuckles Github disassemblies. There are instructions how to do so on Sonic Retro.
It's a good idea to have the disassemblies relatively close to your Sonic3AIR saved data folder, as you have to reference files in the disassemblies from your mod using relative file paths (see below). You can just place them right inside the saved data folder - it won't bother the game anyway.

Once the SonLVL setup is done, you can start editing in SonLVL.
Whenever you save, it **overwrites** certain files in the disassemblies (using example paths for Angel Island Act 1, the others should be self-explanatory):

- Level layout in "skdisasm-master/Levels/AIZ/Layout/1.bin"
- Object placements in "skdisasm-master/Levels/AIZ/Object Pos/1.bin"
- Ring placements in "skdisasm-master/Levels/AIZ/Ring Pos/1.bin"

## Setup your mod

If you haven't done so already, **setup your mod** as described under "How to setup the basic structure of a mod" in the General "Information section" above.

Inside your mod folder, i.e. next to the **"mod.json"** file, add a folder named **"rawdata"** and create a JSON file with an arbitrary name, but ending with ".json". Or have a look at the sample level mod that you can find inside "sample-mods" next to this document.

Sample contents of this raw data JSON file:

```
{
    "icz2_layout":  { "File": "../level/zone05_icz/act2_layout.bin" },
    "icz2_objects": { "File": "../level/zone05_icz/act2_objects.bin" },
    "icz2_rings":   { "File": "../level/zone05_icz/act2_rings.bin" }
}
```

In here, you have to define references to all individual binary files that will make up your level mod. This includes:

- The **layout** for each level you want to mod
- **Objects** in the level
- **Rings** in the level

For each such entry, there is a key string, and a relative file path pointing to one of the binary files that SonLVL creates on saving.
Fill these in correctly for the game to use your changed level layout / objects / rings when you start a level. Note that changes to the JSON file require a **reload of modded data** with F10, and a **restart of the level** (Pause Menu → Restart Act).

## Details on the raw data JSON entries

Inside the JSON file, entries have a format like:
`"aiz1_layout": { "File": "../level/zone01_aiz/act1_layout.bin" }`

The key (here: `"aiz1_layout"`) follows a fixed format, namely:

- First part of the key `"aiz1_layout"` must be the **zone's short name** (`"aiz", "hcz", "mgz", "cnz", "icz", "lbz", "mhz", "fbz", "soz", "lrz", "hpz", "ssz", "dez"`), followed by the **act number** 1 or 2.
- The last part of the key is either `"_layout"`, `"_objects"` or `"_rings"`, depending on what's the meaning of the binary file.

The file path after `"File"` is meant to be the location of the binary file, as a relative path from the JSON file itself (not from the mod folder).

- While working on your mod, it's certainly a good idea to use file references directly into the disassemblies directory, so you can make changes in SonLVL and don't need to copy the binary files around each time.
  E.g.: `"aiz1_layout": { "File": "../../../skdisasm-master/Levels/AIZ/Layout/1.bin" }`
- When you want to distribute your mod, you have to move the binary files somewhere into the mod folder, and change the JSON entry accordingly.
  E.g.: `"aiz1_layout": { "File": "../levels/zone01_aiz/aiz1_layout.bin" }`
  For details on that, see section "Note on distribution" below.

## Can you show me?

No problem. There's a **video** where I went through the steps to make a level mod.
Unfortunately, it still uses the old format with binary file references inside the "mod.json" instead of a JSON file in the "rawdata" folder, but it should give you an idea of how things work.

## Note on distribution

When your mod is ready, you need to manually **move all the .bin files** referenced in the raw data JSON file from the disassemblies into your mod folder. This will most likely require you to do some renamings and/or create subfolders for the files. Feel free to do so as you like – I'd recommend at least making a subfolder "levels" in the mod's base folder and subfolders in there for each zone / act.

Afterwards, remember to change the paths in the raw data JSON file accordingly. The lines will now be something like this:
`"aiz1_layout":  { "File": "../levels/zone1_aiz/act1_layout.bin" }`

# Sprite Modding

## The starting point

In the folder "modding" next to this document you find the "sprites" subfolder, which contains ripped sprite sheets from Sonic 3 & Knuckles. These are the recommended starting points if you want to replace character sprites.
Note that making changes to them will not affect the game, they are just templates for you to copy.

To get started:

- First **setup your mod** as described under "How to setup the basic structure of a mod" in the General "Information section" above.
- Add a subfolder **"sprites"** inside your mod.
- Then **copy** one or multiple of the JSON files and the respective template files into the "sprites" folder. Now you can make changes to both of them as described below.

## The sprite sheet

There are two types of sprite sheets:

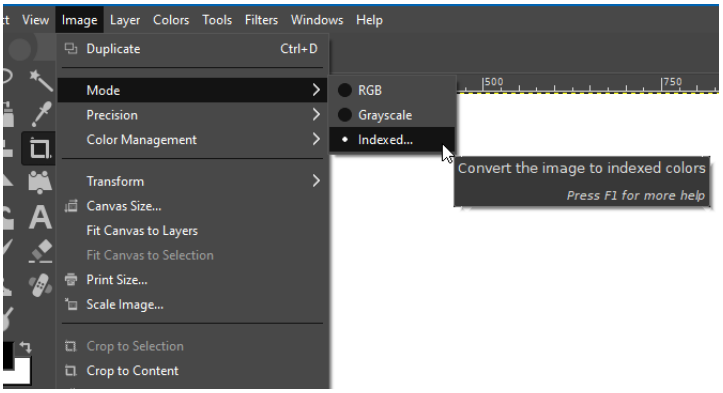
- BMP for palette-based sprites (e.g. character sprites)
- PNG for other sprites (e.g. certain HUD sprites)

The template files will tell you which of these sprite sheet types is best used for which kind of sprites.

**BMP sprites**

Open the BMP file in an image editor application that supports **indexed 8-bit BMPs** and make your changes.

GIMP is just fine for this.
See the screenshot to the right – this is where you find the option to switch to indexed mode. Select 256 colors (or less) in the following menu.




When saving, make sure to save the file with the following settings:

- Windows Bitmap BMP
- Uncompressed / no encoding (in case the editor asks for that)

Please note that not all image editors support indexed images at all.
For instance, Paint.NET does not and thus is not useful for making BMP sprites.

**Important notes on palette-based sprite editing:**

- While editing, only use the first 16 palette indices, and keep in mind that index 0 is reserved for fully transparent pixels.
  There's an exception to this: When using palette modding as well (see below), you can use up to 32 palette indices. In this case, indices 0 and 16 are both reserved for fully transparent pixels.
- In any case, changes to the color palette in the BMP file itself will get ignored. The game will instead use the default character palettes, or modded palettes if present.

- Also make sure that the image editor does not mess up the palette by itself, e.g. resort the colors on export. In case of issues in-game, better double-check by re-importing your BMP into the image editor and view the color palette there.

**PNG sprites**

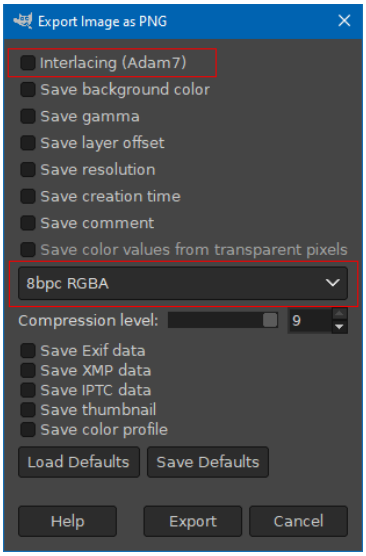
Certain parts of the in-game HUD can be modded, too.
These are not using palettes at all, which makes things easier: What you define in the sprite sheet is what you get in the game.

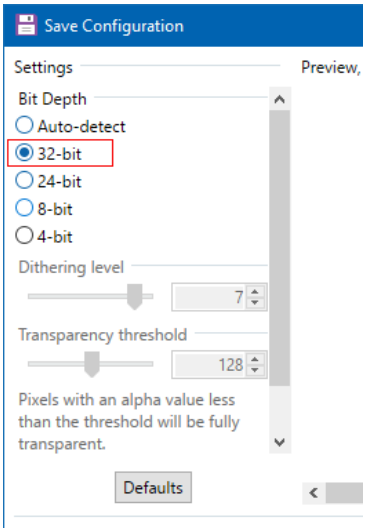
When saving, make sure to save the file with the following settings:

- PNG format
- RGB or RGBA with 8 bits per channel (16 bits are not supported)
- No interlacing

The image to the right shows what the right settings look like in GIMP, with the important parts highlighted.

For PNG sprites specifically, it's okay to use Paint.NET if you like.
See the second image to the right.




**Sprite sizes**

It might be necessary to replace the sprites with larger ones. In that case, you will need to edit the sprite's rectangle inside the sprite sheet as well in the JSON file – see below.

## The JSON file

Inside the JSON, you will find one line per sprite, defining multiple properties:

- The key must be one of certain **predefined keys** that tell the game which sprite of which character this line is about. This is something like `"character_sonic_0x01"`:
  - The first part is one of `"character_sonic"`, `"character_tails"`, `"character_knuckles"` and `"character_supersonic_0x01"` for Super/Hyper Sonic's sprites.
  - The second part is a hexadecimal representation of the sprite index, as used by the game; it's the same as in original Sonic 3 & Knuckles, and ranges from 0x01 to 0xfa at maximum.
- `"File"` is a reference to the **BMP sprite sheet** that this sprite is part of; must be in the same folder as the JSON
- `"Rect"` is the **rectangle** inside the sprite sheet, from the upper left corner in the format `"left,top,width,height"`
- `"Center"` (optional) sets a **center position** inside the sprite; for characters, this is where the character's center is supposed to be, which is not necessarily the sprite's actual center position

Some additional note:

- It does not matter how your JSON files are named. The game simply loads all JSONs it can find in the "sprites" folder.

## Limitations

- As denoted above, sprites are bound to the game‘s character palettes.
- Sprite modding currently only supports replacement of existing sprites, but no new sprites or animations can be added.

# Palette Modding

## What can be modded?

Sonic 3 A.I.R. uses the same palettes for in-game graphics as the original Sonic 3 & Knuckles, with one exception: Sonic, Tails and Knuckles are making use of their very own palettes that don't get shared with other objects any more.
These are the palettes you can modify. This is particularly useful in combination with your own custom character sprites.

## Character palette limitations

Character palettes are limited to 32 color indices, of which 30 can be freely used – indices 0 and 16 are reserved for fully transparent pixels.
You can define independent palettes for all 3 characters, i.e. Tails does not share his palette with Sonic any more.

The game has some limitations for colors in general, namely it uses only 5 bits per color channel. This allows for a total of 2^15 = 32768 possible colors.
(For comparison: The original game uses 3 bits per channel, i.e. 512 possible colors.)
That means, there might be subtle differences between the colors you define in your palettes and how characters look in the game.

## How to define your custom palette

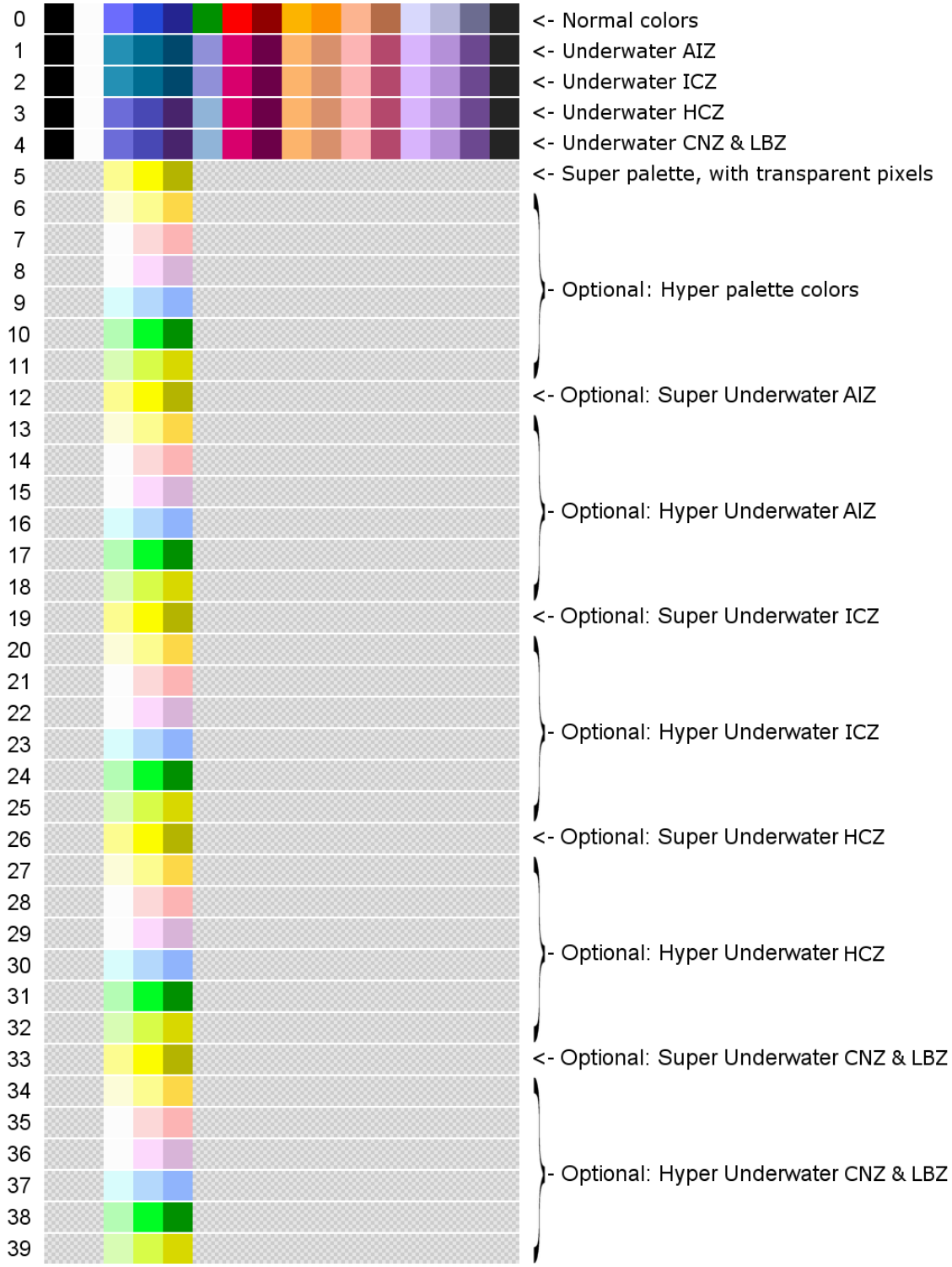
Create file structure in your mod:

- First make sure you have a mod created already following the **usual mod structure** as described under "How to setup the basic structure of a mod".
- Add a subfolder "palettes" to your mod.
- Place one or more PNG image files named "character_palette_sonic.png", "character_palette_tails.png" and "character_palette_knuckles.png" in there.
- You can use the ones inside "modding/palettes" as a string point, these are representing the original character palettes.

Edit a palette file:

- **Each pixel** in the palette PNG image represents **one color** of the palette.
- You may use up to 32 pixels of width, for character palettes with 32 colors. If you only need the original 16 colors, you can leave the image width at 16 pixels. Both is fine.
- There are multiple **pixel lines** for the **different palettes**:
  - Line 0: Normal character palette
  - Line 1: Underwater palette used in AIZ
  - Line 2: Underwater palette used in ICZ
  - Line 3: Underwater palette used in HCZ
  - Line 4: Underwater palette used in CNZ and LBZ
    (note that in the original, certain underwater palettes for Sonic and Tails are identical)
  - Line 5: Super form palette - with a specialty: use fully transparent pixels for all palette colors that should not change or flash in Super/Hyper forms
  - Lines 6 – 11: Hyper form palette cycle for Hyper Sonic

- Lines 12 – 39: Same as lines 5 – 11, but for the different zones' underwater palettes




Save a palette file:

- PNG format
- RGB or RGBA with 8 bits per channel (16 bits are not supported)
- No interlacing

# Raw Data Modding

## What is this about?

The Sonic 3 & Knuckles ROM is the primary data source for Sonic 3 A.I.R., but you can't make changes to it directly - this would interfere with the integrity check the game does to make sure it has the right ROM.
Instead, use the "raw data" modding mechanism that applies binary data into the ROM data after it got loaded from the ROM file.
One possible use-case for this is making palette or level chunk changes, but in theory you can replace all kinds of data inside the ROM.

> ***But be warned...!***
> *This is worth nothing except you know **exactly** where the data you want to overwrite resides in the ROM.*
> *Which makes this a feature only really usable for experts on S3&K ROM hacking.*

## Mod structure

As an example, have a look at the mod inside "sample-mods/sample-rawdata-mod".
It demonstrates the structure of a mod that uses raw data modding, namely:

- A sub-folder with the fixed name "rawdata"
- Inside, one or more JSON with arbitrary names (they have to use the .json extension, of course)
- Plus one or more binary files with arbitrary names

The JSON is built like this:

```
{
	"some_identifier":
	{
		"File": "rawdatafile01.bin",
		"RomInject": "0x123456"
	},

	"another_identifier":
	{
		"File": "another-binary-file.bin",
		"RomInject": "0x0abcde"
	}

	// and possibly more entries
}
```

Each of the entries in there (like "some_identifier") needs the following two parameters:

- `"File"` is the name of a binary file in the same folder as the JSON file itself.
  Its file contents will get copied into the ROM data from the given ROM address on.
- `"RomInject"` is the hexidecimal address inside the ROM where the data will be written.
  Note that from this address the complete file contents will be written, i.e. the binary file size determines the number of bytes written.

For the binary files, there is no special requirements. Their content is uses 1:1 as binary data to write into the ROM.
You can create binary files for certain purposes using e.g. SonLVL, or in general with a Hex Editor like HxD. Yet again, this requires you to know exactly what you're doing.

*Side note: You might have noticed that Level Modding uses the same basic structure for its JSON files, but without the `"RomInject"` parameter.*

*This is because the game handles both kinds of modding in a very similar way:*
*It loads all JSON files inside the "rawdata" folder, and goes through the entries there. Then it either injects the binary data for an entry directly into the ROM, if an address is set via `"RomInject"`. Or it keeps the binary data until some part of the game logic, like level loading, needs it - in that case, hard-coded keys like `"aiz1_layout"` are used.*

# Script Modding

## What do I need?

To make script mods, it's highly recommended to switch the game to **Dev Mode** as described in the Config settings chapter.

Also make sure to check out engine's **Handbook PDF** file that contains a lot of information and documentation on the Dev mode features and the scripting language in general.

## Make a script mod for development

First create an empty mod using the usual mod structure as described in How to setup the basic structure of a mod. Create a subfolder "scripts" in your mod and a text file "main.lemon" inside it. This will be the starting script the game will search for in your mod.
You can also copy the sample mod in "sample-mods/sample-script-mod" next to this document as a starting point.

In a mod, you can overwrite functions of the original scripts to make the game execute your own script function instead of the original function. To do this, add a new function into your main.lemon [1] using the exact same signature (i.e. parameters and return value) as the function you want to replace.
In case you'd like to execute the original function as well inside your own function, call it with a "base." prefix added. See the sample script mod for an example.

1 - Or if you want to use multiple script files, place it in a custom .lemon file and make sure to reference it in the main.lemon via an include line.

## Using Custom Mod Settings

It's possible to add custom settings for your mod to the game's Options menu. They will appear in the leftmost tab of the Options menu, at least if your mod is activated in the Mods menu.

To add settings, you first need to define a **global variable** for each setting somewhere in your script mod like this:

```
global u8 my_own_setting
```

As data type, use one of the following: u8, s8, u16, s16, u32, s32.
Note that is really must be a global variable, a define will **not** work.

The second step is adding a **"Settings"** section into your mod.json file, next to the "Metadata" section, using the following structure:

```
"Settings":
{
	"List":
	[
		{
			"InternalName": "MyOwnSetting",
			"DisplayName": "My Own Setting",
			"Variable": "my_own_setting",
			"DefaultValue": "0",
			"Options":
			{
				"0": "Disabled",
				"1": "Active",
				"2": "Very Active"
			}
		}
	]
}
```

Some explanation on the properties here:

- "InternalName" is a name for the setting, used e.g. as key for saving the value in the player's settings.json file. It's recommended to only use alphanumeric characters and no spaces in that name.
- "DisplayName" is what the setting will be called in the game's Options menu.
- "Variable" is the name of the variable you added in the scripts. This must be the exact same name.
- "Options" lists all possible values and their displayed texts to be shown in the Options menu. You can use either decimal values here, or hexadecimals like "0x18" (same for the default value).

Plus there's these two optional properties you can use as well:

- "DefaultValue" is the initial value for the setting.
- "Category" is the display name of a category for that setting, in case you want to group your settings into multiple categories in the Options menu.

You can define more than one setting by adding multiple blocks into the "List" array.
See the "doc/sample-script-mod" for an example.

## Game-specific Script Functions

In the **Oxygen Engine Handbook PDF**, you can find a list of pre-defined functions that can be used in scripts. However, there's a few more functions that are not listed there because they are only available inside Sonic 3 A.I.R. itself, not in the OxygenApp.

**Discord Integration**

Game.setDiscordDetails(u64 text)

- Overrides the "Details" text in Discord Rich Presence.
- This is the first of two custom text lines, and usually displays the current game mode.

Game.setDiscordState(u64 text)

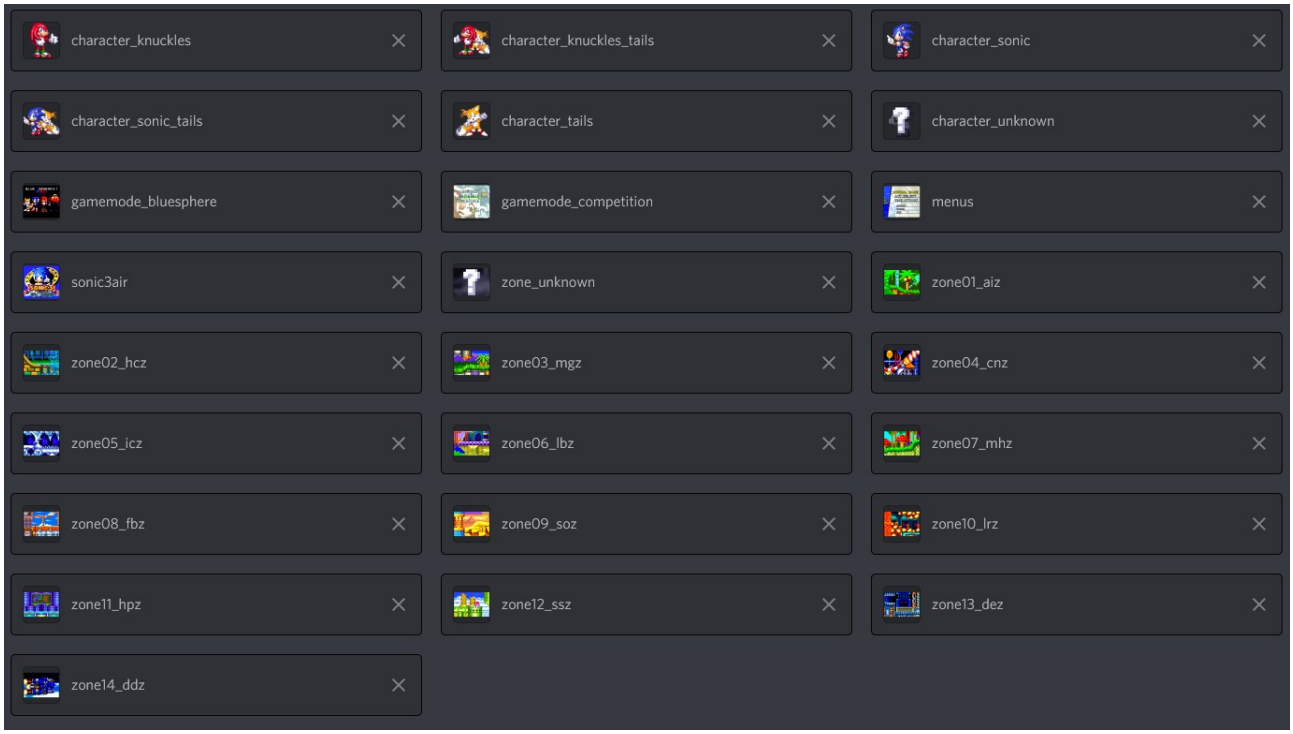
- Overrides the "State" text in Discord Rich Presence.
- This is the second of two custom text lines, and usually displays additional info like the current character selection.

Game.setDiscordLargeImage(u64 imageName)

- Overrides the main image in Discord Rich Presence.
- Due to restrictions set by Discord, you can only choose from a range of pre-defined images, as shown below.

Game.setDiscordSmallImage(u64 imageName)

- Overrides the small image in Discord Rich Presence, which is displayed in the lower right corner of the main image.
- Due to restrictions set by Discord, you can only choose from a range of pre-defined images, as shown below.



# Font Modding

## About font modding

Font modding allows for adding your own custom fonts to the game, for usage with script functions like `Renderer.drawText`.
You can also overload the main game's original fonts, which affects the hard-coded menus.

## Font mod layout

There's a sample mod located at "doc/sample-mods/sample-font-mod" that shows the basic setup of a font mod. You can use that one as a basis, or extend your existing mod in the same way as shown there.

The important new part is the "font" subfolder, which contains one or multiple font definitions.
Each font definition consists of:

- A PNG image similar to a sprite sheet, i.e. it contains all the individual image for all text characters that the font supports. Note that there's currently no support for palette-based fonts, using BMPs as images.
- A JSON text file with basic information about the font, and the locations of each character in the PNG image.

Both files should share the same file name (except for the file extension, obviously).

## Font JSON content

The font JSON looks like this:

```
"ascender": "10",
"descender": "3",
"lineheight": "14",
"space": "1",
"texture": "oxyfont_regular.png",
"characters":
{
	" ": "1 1 3 10",
	"A": "5 1 8 10",
	"a": "redirect: A"
}
```

With the following keys:

- "ascender" - the maximum number of pixels above the base line that characters of your font use
- "descender" - the maximum number of pixels below the base line that characters of your font use
- "lineheight" - total vertical offset between lines of text in pixels; this is usually the ascender plus descender plus a bit of extra space between lines
- "space" - horizontal space in pixels to be added between characters
- "texture" - file name of the image, including the extension ".png"
- "characters" - a list of character definitions

For character definitions:
They always have the character as their key in the JSON.
Their value follows one of these formats:

- A set of four integers in the form `"left top width height"` describing the rectangular position of the character in the font image
- Alternatively, a character can be a redirect, i.e. reuse the image of another character; you'd write `"redirect: A"` in that case, replacing `A` with the character to redirect to

## Using your font

First, you need to know that the game will use your font JSON's file name (without the extension) as the font name.

There's two ways of using your font:

1) In combination with scripts, primarily by using the `Renderer.drawText` function. The first parameter there is the font name, and can be a original game font or your modded font.

2) As a replacement for an original game font (or even the font of another mod). This also affects the otherwise unmoddable menus, if you replace one of the fonts used there.
To replace another font, just give your font the exact same name.

For an overview of the existing original game fonts, have a look at the Sonic 3 A.I.R. Github.
These are also good examples of how font definitions generally look like, as they use the same format as modded fonts.