QR Code Generator with Advanced Styling

This Python script is designed to generate highly customized QR codes from a collection of background images. It leverages the `qrcode` and `Pillow` libraries to offer extensive control over the visual appearance of QR codes, including adaptive coloring, varied module shapes, and customizable finder patterns. The script is multi-threaded for efficient batch processing of numerous image variations.

Table of Contents

---

1. Introduction

`multi_gen.py` is a powerful tool for designers, developers, and enthusiasts who want to create visually distinctive QR codes. Instead of typical monochrome QRs, this script allows you to embed background images and apply a variety of styles, including adaptive coloring that harmonizes with the background, different shapes for data modules and finder patterns, and more. It's particularly useful for generating a large batch of QR code variations for testing or creative exploration.

2. Features

- **Customizable Backgrounds:** Apply background images to your QR codes.
- **Multiple Background Modes:**

    - **Stretched:** Stretches the background image to fill the QR code's background area (excluding outer padding).
    - **Contained:** Resizes the background image to fit within a smaller (approx. 40%) central area of the QR code's background, then centers it.
- **Variable Finder Pattern Shapes:** Choose between `square`, `circle`, or `rounded_square` for the distinctive corner patterns.
- **Dynamic Finder Pattern Colors:**
    - **Static:** Uses fixed black/white.
    - **Dynamic (Single-color):** Attempts to find a bright, contrasting color from the background for the inner ring of the finder pattern, keeping the innermost black.
    - **Dynamic (Multi-color):** Adapts the innermost finder pattern color directly from the background's dominant colors, with an option to reduce its brightness.
- **Finder Pattern Overlay:** Optionally add a semi-transparent colored overlay around the finder patterns to make them stand out.
- **Data Module Shapes:** Render data modules as `square`, `circle`, or `diamond`.
- **Adaptive Data Module Colors:** Data modules can adapt their color based on the underlying pixel color of the background image, darkening or lightening themselves for contrast.
- **Diamond Module Borders:** For `diamond` shape, an optional border can be drawn.
- **Multi-threaded Processing:** Efficiently generates numerous QR code variations concurrently using multiple CPU cores.
- **Comprehensive Output Naming:** Output filenames include tags reflecting the applied styling options, making it easy to identify variations.
- **SVG Support:** Can use SVG files as background images if `CairoSVG` is installed.

3. Dependencies

Before running the script, you need to install the required Python libraries.

- **Pillow:** The friendly PIL (Python Imaging Library) fork, used for image manipulation.
- **qrcode:** A pure Python QR Code generator.

Optional:

- **CairoSVG:** Required for processing SVG background images. If not installed, SVG files will be skipped.

4. Installation

1. **Install Python:** Ensure you have Python 3.7+ installed on your system.
2. **Create a Virtual Environment (Recommended):**

```
python -m venv qr_env
source qr_env/bin/activate # On Windows: qr_env\Scripts\activate
```

3. **Install Libraries:**

```
pip install Pillow qrcode[pil]
pip install cairosvg # Optional: Only if you plan to use SVG background images
```

4. **Save the Script:** Save the provided `multi_gen.py` content to a file named `multi_gen.py` in your desired project directory.

5. Usage Guide

5.1. Preparation

1. **Create an `images` directory:** In the same directory where you saved `multi_gen.py`, create a new folder named `images`.
2. **Place Background Images:** Put all the image files you want to use as QR code backgrounds into the `images` directory. Supported formats include `.png`, `.jpg`, `.jpeg`, `.bmp`, `.gif`, `.tiff`, `.webp`, and optionally `.svg`.
3. **Create an `outputs` directory:** The script will automatically create an `outputs` folder if it doesn't exist. This is where all the generated QR codes will be saved.

Your directory structure should look like this:

```
your_project_folder/
├── multi_gen.py
├── images/
│   ├── background1.png
│   ├── background2.jpg
│   └── another_bg.svg (if cairosvg installed)
└── outputs/ (will be created by the script)
```

5.2. Running the Script

Open your terminal or command prompt, navigate to the `your_project_folder`, and run the script:

```
python multi_gen.py
```

The script will begin processing the images and print progress to the console. It will inform you about the number of worker threads used and the current image being processed.

5.3. Understanding the Output

The generated QR code images will be saved in the `outputs` directory. Each output filename is constructed to reflect the specific combination of settings used for that QR code, following this pattern:

```
<original_filename_base>_<BG_MODE_TAG>_<FINDER_SHAPE_TAG>_<COLOR_MODE_TAG>
<SUBMODE_TAG><DIM_TAG><DATA_SHAPE_TAG><DATA_COLOR_TAG><OVERLAY_TAG><BG_PAD_TAG>.png
```

Example: `nature_Stch_squa_dyna-sc_diaD_adap_over_bgp60.png`

Let's break down the tags:

- **`<original_filename_base>`:** The name of your original background image (e.g., `nature` from `nature.jpg`).
- **`<BG_MODE_TAG>`:**
  - `Stch`: `Stretched` background image mode.

    - `Cont`: `Contained` background image mode.
- `<FINDER_SHAPE_TAG>`:
    - `circ`: `circle` finder pattern shape.
    - `squa`: `square` finder pattern shape.
    - `roun`: `rounded_square` finder pattern shape.
- `<COLOR_MODE_TAG>`:
    - `stat`: `static` finder color mode.
    - `dyna`: `dynamic` finder color mode.
- `<SUBMODE_TAG>` **(only for `dynamic` color mode):**
    - `-sc`: `single-color` dynamic submode.
    - `-mc`: `multi-color` dynamic submode.
- `<DIM_TAG>` **(only for `dynamic multi-color` submode if `REDUCE_INNERMOST_BRIGHTNESS` is True):**
    - `-dim`: Innermost brightness reduced.
- `<DATA_SHAPE_TAG>`:
    - `_diaD`: `diamond` data module shape.
    - `_squD`: `square` data module shape.
    - `_cirD`: `circle` data module shape.
- `<DATA_COLOR_TAG>`:
    - `_adap`: `adaptive` data module color mode.
    - `_statD`: `static` data module color mode.
- `<OVERLAY_TAG>`:
    - `_over`: Finder pattern overlay is enabled. (Absent if disabled).
- `<BG_PAD_TAG>`:
    - `_bgp<value>`: Background padding in pixels (e.g., `_bgp60`). (Absent if `BACKGROUND_PADDING_PX` is 0).

6. Configuration & Customization

The script's behavior is controlled by several constants defined at the top of the `multi_gen.py` file. You can edit these values directly to change the generated QR codes.

6.1. Global Test Variables (Which QRs are Generated)

These lists define the permutations of styles that the script will generate for each input image. By modifying these, you control the *range* of QR codes created.

```
BACKGROUND_MODES_TO_TEST = ["Stretched", "Contained"]
FINDER_SHAPES_TO_TEST = ["circle", "square", "rounded_square"]
FINDER_COLOR_MODES_TO_TEST = ["static", "dynamic"]
FINDER_DYNAMIC_SUBMODES_TO_TEST = ["single-color", "multi-color"]
```

- `BACKGROUND_MODES_TO_TEST`: List of background image application modes to test. Options: `"Stretched"`, `"Contained"`.
- `FINDER_SHAPES_TO_TEST`: List of finder pattern shapes to test. Options: `"circle"`, `"square"`, `"rounded_square"`.
- `FINDER_COLOR_MODES_TO_TEST`: List of finder pattern color modes to test. Options: `"static"`, `"dynamic"`.
- `FINDER_DYNAMIC_SUBMODES_TO_TEST`: List of sub-modes for `dynamic` finder color mode. Options: `"single-color"`, `"multi-color"`.

6.2. Styling & Functional Constants

These constants define the default appearance and behavior of the QR code generation.

6.2.1. General QR & Module Sizing

Constant	Type	Default Value	Description
`DEFAULT_DATA`	`str`	`"https://www.example.com"`	The data string encoded in all generated QR codes. **Modify this to change the QR code's content.**
`DEFAULT_BOX_SIZE`	`int`	`25`	The size in pixels of each QR code module (a "pixel" in the QR matrix). Larger values result in larger QR codes.
`DEFAULT_BORDER`	`int`	`4`	The number of modules forming the white "quiet zone" border around the QR code. QR standard recommends 4 modules.
`DEFAULT_PADDING`	`int`	`4`	Inner padding in pixels between the edge of a module's allotted `box_size` area and the actual drawn shape of the data module (square, circle, or diamond). If 0, the shapes fill the module. If too large, shapes become too small or disappear.
`DEFAULT_ERROR_CORRECTION`	`int`	`qrcode.constants.ERROR_CORRECT_H`	Error correction level. Higher levels (L, M, Q, H) allow for more damage/obscurity before the QR code becomes unreadable, but also make the QR code larger. H is the highest (approx. 30% correctable).

Constant	Type	Default Value	Description
DEFAULT_DARK_MODULE_COLOR	tuple	(0, 0, 0, 240)	Default color (RGBA) for dark data modules when DATA_MODULE_COLOR_MODE is static. The alpha channel controls transparency (0=fully transparent, 255=fully opaque).
DEFAULT_LIGHT_MODULE_COLOR	tuple	(255, 255, 255, 240)	Default color (RGBA) for light data modules when DATA_MODULE_COLOR_MODE is static.
DEFAULT_BACKGROUND_ALPHA	int	255	Global alpha (transparency) applied to the entire background image layer (0-255). 0 is fully transparent, 255 is fully opaque. This affects the background image's overall visibility.

6.2.2. Background Image Application

Constant	Type	Default Value	Description
DEFAULT_BACKGROUND_IMAGE_MODE	str	"Stretched"	Default mode for applying the background image. Can be "Stretched" or "Contained". This sets the default, but is overridden by BACKGROUND_MODES_TO_TEST.
BACKGROUND_PADDING_PX	int	60	Padding in pixels around the entire QR code area, where the background image will *not* be applied. This area will be filled with a solid white color (or whatever the initial canvas background is).

6.2.3. Finder Pattern Styling

| Constant | Type | Default Value | Description | | DEFAULT_FINDER_PATTERN_SHAPE | str | "rounded_square" | Default shape for the finder patterns. Can be "square", "circle", or "rounded_square". This sets the default, but is overridden by FINDER_SHAPES_TO_TEST. | | ROUNDED_RADIUS_FACTOR | float | 1.2 | For rounded_square finder patterns and alignment patterns, this value (multiplied by box_size) determines the corner radius in pixels. A value of 1.0 means the radius is equal to the module size. | | REDUCE_INNERMOST_BRIGHTNESS | bool | True | If True, the inner ring of the finder pattern (when in dynamic/multi-color mode) will have its brightness reduced by INNERMOST_BRIGHTNESS_REDUCTION. | RED| REDUCE_INNERMOST_BRIGHTNESS | bool | True | If True, the innermost color of the finder pattern (when using dynamic mode and multi-color submode) will have its brightness reduced

by `INNERMOST_BRIGHTNESS_REDUCTION`. This helps the innermost block stand out against a potentially bright background color. |

6.2.4. Data Module Styling

Constant	Type	Default Value	Description
`DATA_MODULE_SHAPE`	`str`	`"diamond"`	Default shape for individual data modules. Options: `"square"`, `"circle"`, `"diamond"`. This sets the default, but is combined with test parameters.
`DATA_MODULE_COLOR_MODE`	`str`	`"adaptive"`	Default color mode for data modules. Options: `"static"`, `"adaptive"`. This sets the default, but is combined with test parameters.
`DEFAULT_DIAMOND_BORDER_WIDTH`	`int`	`0`	If `DATA_MODULE_SHAPE` is `"diamond"`, this specifies the width in pixels of an optional border drawn around the diamond. Set to 0 for no border. The border color is the contrasting color to the fill color.
`INNERMOST_BRIGHTNESS_REDUCTION`	`float`	`0.25`	If `REDUCE_INNERMOST_BRIGHTNESS` is `True`, this is the factor by which the brightness of the innermost finder pattern color is reduced (0.0 = no reduction, 1.0 = full reduction to black).

6.2.5. Color Processing Thresholds

Constant	Type	Default Value	Description
`ADAPTIVE_COLOR_RADIUS`	`int`	`10`	For `adaptive` data module coloring, this defines the radius in pixels around the module center where the script samples colors from the background image to determine the dominant color.
`ADAPTIVE_LUMINANCE_THRESHOLD`	`int`	`130`	HSL Luminance threshold (0-240 scale). Used in `adaptive` data module coloring and `dynamic` finder pattern coloring. Colors below this are considered "dark", above are "light."
`ENABLE_FINDER_OVERLAY`	`bool`	`True`	If `True`, a semi-transparent colored layer will be drawn around the finder patterns.

6.2.6. Debug Flags

Constant	Type	Default Value	Description

Constant	Type	Default Value	Description
`DEBUG_CONTAINED_MODE`	`bool`	`False`	If `True`, enables verbose print statements to the console about the calculations for the `Contained` background image mode. Useful for debugging layout issues with `Contained` mode. Requires `print_lock` to be passed to `create_qr_code` (which it already is).
`SVG_SUPPORT`	`bool`	`True`	Automatically set to `True` if `cairosvg` is successfully imported, `False` otherwise. Controls whether SVG files are processed. If `cairosvg` is not installed, setting this manually to `True` will still result in `False` due to the `try-except` block, but you could theoretically force it if you had a non-standard SVG renderer (though not supported out-of-the-box by this script).

7. How It Works (Technical Deep Dive)

The script orchestrates several components to produce the stylized QR codes.

7.1. Multi-threading Architecture

The script uses a producer-consumer model with Python's `threading` and `queue` modules:

1. **Task Generation (Main Thread):** The main part of the script iterates through all combinations of input images and desired styling parameters (from `*_TO_TEST` lists). For each combination, it constructs a task tuple (`input_filepath`, `output_qr_code_path`, `filename`, `settings_dict`) and places it into `task_queue`.
2. **Worker Threads:** A pool of `num_worker_threads` (defaulting to CPU count - 1) are spawned. Each worker thread runs the `process_image_worker` function.
3. **Task Consumption:** Each worker thread continuously fetches tasks from `task_queue`.
4. **QR Generation:** For each task, the worker calls the `create_qr_code` function with the specified parameters.
5. **Result Reporting:** After processing (or error), the worker puts a result dictionary (`{"status": ..., "filename": ..., "output_path": ...}`) into `result_queue`.
6. **Synchronization:**
    - `task_queue.join()`: The main thread waits for all tasks in `task_queue` to be marked as done, ensuring all QR codes are processed before summarization.
    - `print_lock`: A `threading.Lock` is used to prevent multiple threads from writing to the console simultaneously, which would result in garbled output. All `print()` statements within the worker function are protected by `with print_lock:`.

7.2. QR Code Generation Core

The script uses the `qrcode` library to generate the base QR code matrix.

- `qrcode.QRCode(...)`: Initializes a QR code object with parameters like `box_size`, `border`, and `error_correction`.
- `qr.add_data(data)`: Adds the desired string data.
- `qr.make(fit=True)`: Generates the QR code matrix, adjusting version if needed.
- `qr.modules`: This 2D boolean array represents the QR code's black (`True`) and white (`False`) modules.
- `qr.modules_count`: The dimension of the QR matrix (e.g., 21 for version 1, 25 for version 2).
- `qr.border`: The actual border modules used after `fit=True`.

- `qr.version`: The determined QR code version.
- `qrcode.util.pattern_position(version)`: Used to determine the center coordinates of alignment patterns for higher QR versions.

7.3. Background Image Processing

1. **Loading:** The background image is loaded using `PIL.Image.open()`. If the file is an SVG and `cairosvg` is installed, `cairosvg.svg2png()` is used to convert the SVG to a PNG in memory, which is then opened by Pillow.
2. **Canvas Setup:** A new `RGBA` Pillow image (`final_image`) is created to serve as the base canvas for the entire QR code, including its quiet zone. An intermediate `background_canvas` is used for applying the background image to avoid alpha blending issues.
3. **Padding:** The `BACKGROUND_PADDING_PX` defines a border around the entire QR code. The background image is only applied *within* this padded area.
4. **Mode Application:**
    - **"Stretched":** The background image is simply resized (using `Image.Resampling.LANCZOS` for quality) to fit the available space within the padded area and pasted onto the `background_canvas`.
    - **"Contained":** This mode aims to fit the background image into a central portion of the QR code (specifically, a square that covers approximately 40% of the *padded area's total area*), while maintaining its aspect ratio. The image is resized to fit this "target contained box", and then centered within the entire padded area. This prevents the background from being stretched across the entire QR code, making it more visible in the center.
5. **Global Alpha:** The `DEFAULT_BACKGROUND_ALPHA` is applied to the `background_canvas` using `Image.alpha_composite` to control the overall transparency of the background image.

7.4. Color Extraction and Adaptation

The script uses several functions to intelligently select colors based on the background image:

- **`get_dominant_colors(img, ...)`:** This function takes an image (or a thumbnail thereof) and uses Pillow's `quantize` method to identify the most frequent colors in the image. It's used to get a general palette for the `dynamic` finder pattern modes and as a fallback for `adaptive` data module coloring.
- **`get_prominent_color_in_region(image, center_x, center_y, radius)`:** For `adaptive` data module coloring, this function samples a small circular region around the center of each data module. It quantizes colors within that specific region to find the most prominent color there. This provides highly localized color adaptation.
- **`calculate_hsl_from_rgb(r, g, b)`:** Converts RGB colors to HSL (Hue, Saturation, Luminance). Luminance is crucial for determining if a color is "dark" or "light", which is used for adaptive coloring decisions.
- **`reduce_rgb_brightness(rgb_tuple, factor)` / `increase_rgb_brightness(rgb_tuple, factor)`:** These adjust the brightness of an RGB color. Used in `adaptive` data module coloring and `dynamic` multi-color finder patterns.
- **`get_contrasting_color(dominant_colors_list, type, ...)`:** Attempts to find a dark or light color from the dominant colors of the *entire* background image that meets a certain luminance threshold. This is used as a fallback or default for adaptive colors when the local region is too white/black.
- **`determine_finder_colors(...)`:** This function encapsulates the logic for choosing the outer, inner, and innermost colors of the finder patterns based on the `finder_color_mode` and `finder_dynamic_submode` settings. It leverages the dominant colors from the background image.

7.5. Drawing Logic

The script meticulously draws each part of the QR code onto a `final_image` canvas, using `PIL.ImageDraw`. Drawing is done on temporary `RGBA` layers (e.g., `data_module_layer`, `pattern_layer`) which are then alpha-composited onto the `final_image`. This allows for proper alpha blending and avoids artifacts.

- **Data Modules:** The script iterates through the QR matrix modules. For each module that is *not* a finder or alignment pattern:
  - It determines the fill color based on `DATA_MODULE_COLOR_MODE` (`static` or `adaptive`).
  - It then draws the selected shape (`square`, `circle`, or `diamond`) within the module's allocated `box_size` area, applying `padding`.
  - For `diamond` shapes, it first draws a slightly larger diamond for the `diamond_border_width` (if set) and then draws the main fill diamond on top, creating a border effect.
- **Finder Patterns:** These are the three large square patterns at the corners.
  - `draw_finder_patterns` function handles these.
  - It first optionally draws a `finder_overlay_color` around them if `ENABLE_FINDER_OVERLAY` is `True`.
  - Then, it draws three concentric shapes (outer, inner, innermost) according to the `finder_shape` (`square`, `circle`, `rounded_square`) and the colors determined by `determine_finder_colors`.
- **Alignment Patterns:** These are smaller square patterns found in larger QR versions (versions 2 and up).
  - `draw_alignment_patterns` function handles these.
  - They are drawn similarly to finder patterns, with three concentric shapes and colors derived from the finder pattern's palette.

8. Error Handling

The script includes `try-except` blocks to handle common issues:

- **Image Loading:** `FileNotFoundError`, `UnidentifiedImageError`, `ValueError` (e.g., if SVG without CairoSVG) are caught when loading background images.
- **Image Processing/Drawing:** General `Exception` catches for potential errors during image manipulation and drawing, allowing the script to continue processing other images even if one fails. Warnings are printed to `sys.stderr`.
- **Parameter Validation:** Basic checks are in place for `padding`, `diamond_border_width`, and `background_alpha` to ensure valid ranges and prevent drawing errors due to misconfigured values.

9. Limitations and Considerations

- **Fixed QR Data:** The `DEFAULT_DATA` constant applies to *all* generated QR codes in a batch. If you need different data for different QR codes, you'd need to modify the script to read data from a list or file, or run it multiple times with different `DEFAULT_DATA` values.
- **SVG Support:** While included, `cairosvg` can be a complex dependency to install depending on your system, as it relies on external C libraries (Cairo). If you encounter issues, you can stick to raster image formats.
- **Performance:** While multi-threaded, processing a very large number of high-resolution images with many variations can still be CPU and memory intensive. Monitor your system resources.
- **Pillow Version Compatibility:** Ensure your Pillow version is relatively recent for `rounded_rectangle` and `quantize` methods to work optimally.
- **Quiet Zone Color:** The quiet zone (outer border) is implicitly white because the `final_image` canvas is initialized with white. If you want a different color, you'd need to change the initial canvas creation or draw a rectangle before compositing.

10. Future Enhancements

- **CLI Arguments:** Use `argparse` to allow setting parameters (like `DEFAULT_DATA`, `box_size`, modes to test) directly from the command line without editing the script.
- **Logo Overlay:** Add functionality to embed a logo image in the center of the QR code.
- **More Shapes:** Introduce additional shapes for data modules and patterns (e.g., hexagons, stars).
- **Gradient Colors:** Allow gradient fills for modules or patterns.
- **JSON/YAML Config:** Externalize all configuration constants into a separate config file (e.g., `config.json`), making customization easier without touching the code.
- **Progress Bar:** Implement a visual progress bar for long processing batches.
- **Custom Color Inputs:** Allow specifying static colors directly instead of relying on defaults.
