ChatGPT

High-Quality ASCII Art Banners from Images

Tools and Libraries for Image-to-ASCII Conversion

To create ASCII art banners from images, several tools and libraries are available:

- **jp2a:** A popular CLI tool that converts JPEG images to ASCII. It supports custom character sets, inversion for light backgrounds, and even ANSI color output [1]. For example, running `jp2a --colors image.jpg` will produce a color ASCII rendering in the terminal [2] [3]. (Note: jp2a only natively reads JPEG; other formats may need conversion to JPEG first.)
- **ascii-image-converter:** A modern cross-platform CLI tool (by TheZoraiz) that handles many image formats (even GIFs) and outputs ASCII art to the console [4]. It offers features like color (`-C` for truecolor or 256-color), **custom brightness maps** (`-m` to supply your own character string), **"complex" mode** for more detailed character ramps, **resizing flags** (`-W`, `-H`, or `-d` to set dimensions), and even a **braille mode** for higher resolution [5] [6]. This tool is actively maintained and was recommended as a robust solution [7].
- **Chafa:** A powerful ASCII/ANSI art generator that can use **extended Unicode symbols** (blocks, braille, etc.) for better fidelity. It's actively developed and very configurable [4]. Chafa supports full-color output and can target pure ASCII if needed (`--symbols ascii`). Its default behavior uses a wide range of Unicode symbols to improve quality beyond simple ASCII [8]. It can even output true images via sixel or Kitty protocols [9], making it extremely versatile for terminal graphics.
- **CACA/`img2txt`:** The *libcaca* library and its `img2txt` utility convert images into colored ASCII text. They support common formats and produce ANSI-colored output in the terminal [10]. This is an older solution but still works (it's the backend for many ASCII image viewers).
- **AAlib/`aview`:** Another classic approach using AAlib (ASCII Art library). Tools like `asciiview`/`aview` render images as ASCII (monochrome by default). These may require converting images to PNM format and are less flexible (mainly historical interest) [11].
- **Online/Web Tools:** There are web-based converters (e.g. asciify.art) and libraries in various languages (Python's PIL-based scripts, Node's `asciify-image` package, etc.), but for command-line usage in a terminal, the above native tools are most practical.

Tip: Many tools support **outputting HTML** as well, which can preserve coloring in a browser [12]. But for a terminal banner, you'll typically output plain text or ANSI-colored text directly to the console.

Image Preparation Best Practices

Getting a good ASCII banner starts with prepping the source image:

- **Choose a Simple, High-Contrast Image:** Images with bold shapes and clear contrast translate best. Photographs should have strong lighting or distinct edges; logos or icons often yield superior results [13]. Busy, highly detailed images can turn into an unrecognizable jumble of characters [14]. Consider using a **simplified or vector version** of a logo rather than a full photo for clarity.

- **Crop and Focus on the Subject:** Remove unnecessary background or details and make the subject as large as possible in the frame. This maximizes the meaningful content in the limited ASCII resolution [15]. For example, if your banner area is 80 columns wide, crop the image so the main subject fills that width.
- **Resize to Target Resolution:** Scale the image (preserving aspect ratio) to the final output size (in characters). Most tools let you specify output width/height (e.g. `--width` in jp2a [16] or `-W` in ascii-image-converter [17]). Aim for a width that fits typical terminals (80–120 characters) so it doesn't wrap [18]. For many cases ~80–100 cols wide is a sweet spot: it retains detail but can still be viewed without scrolling [18].
- **Account for Font Aspect Ratio:** Characters are usually taller than they are wide. If your ASCII output looks vertically stretched or squashed, adjust the image's aspect. You might need to **scale the height** of the image (or the output characters) to counteract non-square character cells [19]. For instance, some guides recommend stretching the image vertically ~2× before conversion, since a console font will "squish" it back down [19] [20]. Many tools default to preserving the image's pixel aspect ratio, so you may have to trial-and-error the height or use a tool's aspect override (`--stretch` in jp2a ignores aspect [21]).
- **Convert to Grayscale (for Monochrome ASCII):** Removing color simplifies the mapping to characters. Most converters do this internally (e.g., jp2a's default is grayscale unless color is enabled [22]). Ensure the grayscale image has good contrast – you can tweak brightness/contrast beforehand to emphasize important features.
- **Increase Contrast & Sharpness:** Slightly exaggerating the image's contrast and edge definition can improve the ASCII result [23]. The conversion is essentially a form of quantization, so clearer distinctions in brightness yield a cleaner mapping to distinct characters. You can use image editing tools or flags in converters (like ascii-image-converter's `--negative` or adjusting its brightness levels) to get a crisper look.
- **Dithering for Detail:** If using a very limited palette (e.g., black and white only), applying a dithering filter can simulate grayscale by patterning characters [23]. Some converters have dithering options (e.g. `--dither` with braille mode in ascii-image-converter [24]). Use this carefully – dithering can sometimes introduce noise, but it helps to avoid large flat areas and banding.
- **Background Inversion:** Decide whether your ASCII art will be shown on a dark background (common in terminals) or light background. Many tools have an invert option (jp2a `--invert` [25], Ascii Generator's "Invert Output" button [23]) to flip the mapping. For a dark terminal background, you typically want **dark characters for dark image areas** and spaces (or light punctuation) for highlights. On a white background, the opposite mapping may look better. Setting this correctly ensures the ASCII art "pops" with the correct polarity of brightness.

Character Sets and Brightness Mapping

At the heart of ASCII conversion is mapping pixel brightness to characters. Characters have varying "visual density" – how much ink they cover on the screen [26]. For example, `@` and `#` are very dense (good for dark areas), while `.` and space are very light (good for bright areas or highlights) [26]. Using a **gradient string** of characters from darkest to lightest is a common technique. A classic simple ramp is:

```
@$#*+=-:.
```

Here `@` represents the darkest pixels and ` ` (space) the brightest (essentially empty). More elaborate ramps exist – one used in many scripts is:

```
@%#*+=-:.
```

The idea is the same: each character in the sequence corresponds to a range of brightness values from black to white. When converting an image, each pixel (or block of pixels) is replaced by a character whose intensity best matches that pixel's grayscale value [27].

Best practices for character sets:

- **Use a Sufficient Range:** Too few characters (e.g. just `@` and space) will lose detail, while too many characters can produce muddled results if the differences are subtle [28]. Around 10–12 levels (as in the example ramp above) is a proven balance for most images [28], but you can experiment. Many tools default to a built-in palette around this size.
- **Order by Intensity:** If you customize the set, always list characters from most solid to most empty [29]. For instance, `--chars="@@##==..  "` in jp2a would be a custom ramp. The converter will map pixel values into equal divisions across this range [28].
- **Consider Character Shapes:** Characters differ not just in density but shape. A block `@` covers most of a cell uniformly, whereas `M` or `#` cover a lot but with some holes. Depending on the font, some characters might actually appear darker than others even if they cover similar area (e.g. `@` vs `$`). The defaults chosen by tools are often empirically determined to look right in a typical monospaced font [26]. If a particular character in your output looks off, you can tweak the set.
- **"Complex" Character Sets:** Some advanced modes use a larger variety of characters (including letters) to try to capture fine differences [6]. This can improve detail **if** the image is large enough, but at smaller sizes it may introduce visual confusion (your eye might start reading letters or seeing patterns). Use complex mode or large palettes only when you have sufficient resolution for them to make a difference, and the result is still recognizable.
- **Unicode Blocks and Braille:** Strictly speaking these go beyond ASCII, but note that many tools can use **block elements (**  **, **  **, **** **etc.) or braille patterns** to achieve a pseudo-"ASCII" art. These symbols can represent finer gradations or multiple pixels per character (braille patterns are 2×4 dot matrices). For example, using Unicode half-block `▄` can double the vertical resolution (two pixels per text row) [8]. Such symbols aren't in the standard 95 ASCII characters, but they can dramatically improve quality. If your use-case allows, you might explore these for better detail (see **Terminal Graphics** section below).

Terminal Compatibility (Fonts, Aspect Ratio, and Color)

Creating the ASCII art is half the battle – displaying it correctly in a terminal is the other half. Key considerations include:

- **Monospace Font:** Always view ASCII art in a monospaced font. Proportional fonts will distort the spacing and ruin the image [30]. Most terminals use monospace by default, but if you're embedding in an application, ensure a monospace font is used for the banner. Common choices like Courier, Consolas, or the terminal's default are fine [30]. Also, avoid any font smoothing or weird line spacing that could throw off alignment.

- **Character Cell Aspect Ratio:** As mentioned, the typical terminal character is not a perfect square. For instance, a 8x16 cell means characters are twice as tall as they are wide. This means an image that is *N* pixels tall may need more or fewer rows of text to appear with the correct proportions. Many ASCII tools assume a 2:1 height:width ratio by default (some implicitly, some configurable). If you notice your output looks stretched, you might need to adjust the scaling. One trick is to play with the output dimensions: e.g., if your image is appearing too tall, try reducing the number of rows (or increasing columns) until it "looks right." In the Steam profile ASCII guide, they explicitly stretched the image vertically before conversion to counter the squashing effect of the font [19]. In general, **preserve the image's original aspect ratio** during conversion, but be aware you might need to tweak it for your specific font/terminal.
- **Terminal Width & Wrapping:** Consider the range of terminal widths your banner might be viewed in. If it's for your own setup, you know your column count. But if distributing (say a tool's startup banner), 80 columns is the safe minimum. Some tools can auto-detect or fit to terminal width (e.g. `--term-width` in jp2a [31] or ascii-image-converter's `--full` flag to fill current width [32]). This ensures no unintended line wraps. A wrapped line in ASCII art will misalign and break the image. To be safe, you can also document the recommended width or include a smaller fallback version for narrow screens.
- **ANSI Color Support:** If you plan to use color ASCII art, verify the target terminal supports the necessary color depth. Modern terminals (xterm, GNOME Terminal, Windows Terminal, etc.) usually support 256-color and often 24-bit "TrueColor" via ANSI escape codes. Older terminals or very minimal setups might only support the basic 16 colors. Tools like jp2a let you specify color depth (`--color-depth=4|8|24`) to target 16, 256, or truecolor [33]. By default, ascii-image-converter will try 24-bit and fall back to 8-bit if needed [5]. Keep in mind that the first 16 ANSI colors can be remapped by user themes (e.g. "bright green" might not actually be bright on some schemes), so if you use a limited palette, test on default schemes or stick to 256 extended colors for predictable results [34] [35].
- **Windows vs. Unix:** Most Linux/macOS terminals handle ANSI codes and Unicode well these days. Windows' old Console (cmd.exe) historically had issues with ANSI colors and Unicode characters, but the newer Windows Terminal and PowerShell Core do support them. If targeting Windows cmd, you might need to enable `ENABLE_VIRTUAL_TERMINAL_PROCESSING` or use a library to handle it. Also, some Unicode characters (like braille or block elements) might not render in certain Windows fonts by default [36]. It's worth testing on the platforms you expect.
- **Line Endings and Spacing:** Ensure that your output uses the correct newlines and doesn't have trailing spaces stripped. Some terminals or editors might trim spaces at end of line, which can ruin an image (because a space is a real "white pixel" in ASCII art). In a programming context, printing each line exactly and preserving spacing is important.

Monochrome vs. Color ASCII Art

Monochrome ASCII Art (no ANSI coloring) uses characters' brightness alone to depict the image. This has a classic look and works in any environment. It's best for emphasizing shape and form, especially if the source image is essentially a silhouette or line art. To maximize clarity in monochrome:

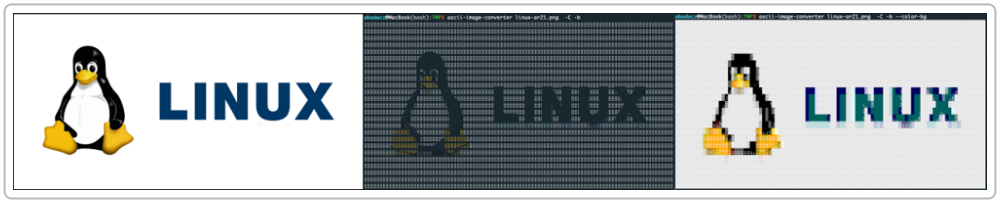
- Use a well-chosen character ramp (as discussed) to create enough contrast in the ASCII output.
- Leverage spacing and solid blocks: e.g. many converters will actually use the space character for the brightest regions, effectively leaving them blank, which on a dark background looks like "white".

Conversely, for a light background you'd want spaces to represent dark regions (hence an invert option) [25].

- If the image has color regions but you're doing monochrome, those regions will all be rendered in the same "ink" – so distinct colors that have similar brightness may merge. Sometimes manually adjusting the source to different grayscale levels for different parts can help the ASCII art distinguish them.

Color ASCII Art integrates ANSI color codes so that each character is not only chosen for brightness but also tinted to match the image's color. This can greatly increase recognizability: for example, a red circle on a white background will appear as red ASCII characters forming a circle – the shape from characters plus the color together convey the image. Some tips for color ASCII:

- **Terminal TrueColor:** Aim to use 24-bit color if possible for a closer match to the original image [5]. If not, 256-color can often suffice (the image will be quantized to the 256-color palette). TrueColor ASCII art can be remarkably detailed in color gradation, though subtle photo-realism is still hard in pure text.
- **Balance Detail with Color:** You might not need as many different characters when using color. Some approaches use a minimal shape set (even just one character like `@` or `█`) and rely on color blocks – this edges into "ANSI art" territory. But with standard ASCII characters, you can sometimes reduce the character palette because color is doing more work. For instance, ascii-image-converter's `--grayscale` vs `--color` options: `--color` prints full colored text, while `--grayscale` uses ANSI but only gray shades [37]. In full color mode, you might stick to a simpler brightness ramp (or even a single brightness like all `@`) since hue and saturation come from color.
- **Colored Background vs. Foreground:** A trick for fidelity is using the background color for pixels. For example, printing a space character with a colored background is effectively a colored rectangle. Tools like ascii-image-converter support `--color-bg` which applies the pixel color to the background and prints a space (or a braille dot) [38]. This can increase the "fill" of color in each character cell, making the ASCII art look more solid and closer to the original image (especially when using braille or half-blocks). The trade-off is you lose some ascii character detail – it becomes more like a mosaic of colored cells.
- **Palette Limiting:** If an image has too many colors, the ASCII output can look chaotic (color noise). Consider reducing the image's color palette first (posterize it) or use a tool's option to limit colors. For instance, ASCII Silhouettify lets you set a max number of colors; setting it to a low number (like 6) can produce a cleaner, posterized ASCII art that still captures the essence of a logo [34] [39]. This is especially useful for logos with a few brand colors.
- **Examples of Color vs Monochrome:** The difference can be stark. In the figure below, the Linux Tux logo is shown in its original form (left), as pure monochrome ASCII, and as colored ASCII art. The colored version preserves the yellow belly and green text, making it much more recognizable at a glance, whereas the monochrome relies solely on brightness [40].



Left: original Linux logo image; Middle: ASCII art in monochrome; Right: ASCII art with ANSI colors (generated by ascii-image-converter) [40].

In summary, use monochrome when you want a stylized or classic look (or need broad compatibility), and use color when fidelity and visual impact are more important (and you know your audience's terminal supports it). Often, color ASCII art makes the "wow" factor for banners, but ensure it still looks good if someone runs your program without color (you might provide a `--no-color` option or detect dumb terminals to fall back to monochrome).

Readability and Visual Impact at Various Sizes

ASCII art should be scalable in the sense that it remains recognizable within the constraints of different terminal sizes. Here are some guidelines to ensure your banner maintains its visual impact:

- **Optimal Width:** Design your ASCII banner for a typical width, but be mindful of smaller windows. If you target 100 columns [18], it will look great on full-width for most, but might wrap on an 80-column display. Targeting 80 columns ensures no wrapping almost anywhere, but you sacrifice some detail. Decide what's acceptable for your use-case. If this is for a personal or controlled environment (like an internal tool where you know the settings), you can go wider. Otherwise, 80 is a safe default, perhaps with an option to generate a more detailed version if the user has a wider terminal.
- **Maintain Aspect Ratio:** As discussed, distortion ruins readability. Always check that the aspect of the ASCII art is correct when displayed. If the image looks squashed or stretched, adjust your approach (either by resizing the input differently or by using the tool's options) until proportions match the original. A quick test is to display a circle or perfect square logo – if your ASCII version looks like an ellipse or rectangle, you need to tweak the scaling.
- **Test at Different Font Sizes:** Uniquely, you can "zoom out" by decreasing font size or zooming out the terminal, which effectively increases the resolution (more characters become visible) for the ASCII art. This can reveal more detail [41]. Encourage users (or yourself) to view the banner without super large font if possible. Conversely, at very small terminal widths or large font, the art might degrade to abstract blobs. Find a balance and perhaps provide multiple sizes.
- **Use Padding Wisely:** Sometimes adding some horizontal or vertical padding (blank lines or spaces) around the ASCII art can help it not to collide with other UI elements and also give it breathing room so it's visually distinct. However, too much whitespace might cause the image to be off-center if the terminal resizes. You can center text banners by padding, but that's tricky across varying widths. A simpler approach is often to left-align and let the user's environment center it if needed.
- **Consider Background Color:** ASCII art is essentially a stencil. If your art uses spaces for highlights, it assumes the terminal background color provides the contrast. If your application might run in terminals with different background colors (some users prefer light backgrounds), consider offering an inverted version or using explicit ANSI background coloring to ensure it looks right. For example, jp2a has a `--background=light` option to optimize the character selection for a light background [42]. You could also detect the terminal's background if possible (some terminfo provide this) and choose the ASCII palette accordingly.
- **Maintain Readability:** Step back and look at the ASCII banner as a whole. Does it clearly convey the image or text it's supposed to? If not, reduce complexity. It's better to have a bold, easily recognizable ASCII logo (even if simplified) than a very detailed but unintelligible block of characters. Sometimes fewer characters (or a manually touched-up ASCII) works better for small sizes. **Keep critical features** – e.g., eyes on a character, or an outline of a shape – emphasized in the ASCII version, even if it means exaggerating them compared to the source.

Finally, remember that ASCII art, by nature, is an **approximation**. It won't be pixel-perfect, so aim for capturing the essence of the image. If viewers can instantly tell what it is, your banner is successful!

Vector Logos vs. Photographs in ASCII

When it comes to source images, **simpler is better** for ASCII conversion. **Vector-style logos or icons usually outperform detailed photographs** as ASCII art banners. Logos often consist of flat colors and clean shapes – these translate to consistent character blocks and clear outlines. In contrast, photographs contain continuous tones, shadows, and fine details that get lost or create visual noise in ASCII form [43].

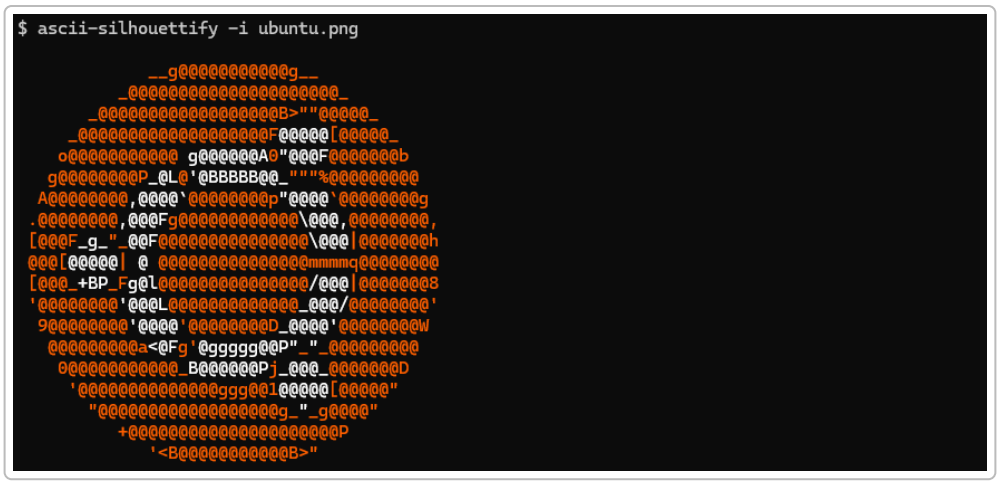
Why logos and icons work better:

- They typically have high contrast (e.g., a dark logo on a light background), which is exactly what ASCII art needs [13]. Each region of the logo can be mapped to a uniform cluster of characters (or colored characters) without too much internal detail needed.
- Logos are designed to be recognizable even at small sizes or low resolutions – that property carries over to ASCII art. The strong shapes or text in a logo will still be evident when "pixellated" into characters.
- You can often preprocess a logo to a **silhouette** or a limited-color version and get great ASCII results. For instance, ASCII Silhouettify (a specialized tool) fills uniformly the regions of a logo with ASCII characters, avoiding gradients altogether [43]. It expects a flat graphic and produces a very clean output, essentially an ASCII "stencil" of the logo.

Challenges with photographs:

- Photographs rely on many shades of color and fine details which ASCII art can only approximate. Unless the photo is high-contrast or iconic (think of the classic ASCII Mona Lisa – it's recognizable mostly because we already know the image), a random photo might just look like speckles in ASCII.
- If you do need to ASCII-ify a photo, consider converting the photo into a **cartoon or posterized version** first – reduce it to fewer colors or clear edges. This moves it closer to a "logo" style for the ASCII conversion. Also focus on key features (for a face, high-contrast lighting to get the outline of facial features, etc.). Some advanced workflows apply edge detection on photos before rendering to ASCII, which can help outline important lines [44].

Bottom line: If you have the option, use a simplified or vector image for your ASCII banner. Many tech projects do this – e.g., an ASCII art company logo – because it yields a much cleaner result than a photograph would. The example below demonstrates this with the Ubuntu logo:



ASCII art banner of a vector-style logo (Ubuntu). Note the clean lines and uniform colored regions – these come from a simplified source image, yielding a more readable ASCII output [43] .

(By contrast, trying to render a complex photo of a landscape in ASCII within the same dimensions might result in a muddled pattern of characters.)

If your goal is maximum clarity and visual impact, lean towards logos/graphics. Photographs can certainly be done in ASCII (and color helps them a lot), but they often require larger outputs and careful tuning to look good.

Beyond ASCII: High-Fidelity Terminal Graphics

Since you mentioned wanting to get *"as close to the image in the CLI as a banner"* as possible, it's worth noting some options outside traditional ASCII art:

- **Unicode "Mosaics":** As mentioned, using Unicode block elements or braille can significantly increase the resolution of your text graphics. Tools like Chafa automatically combine many Unicode symbols to improve quality [8] [45] . This isn't limited to the 95 ASCII chars, but it stays in the realm of text that any terminal can render. The result can be much closer to the original image (at the cost of losing the classic ASCII look). For example, braille dots allow an 8-dot matrix per character cell, and full block `█` characters can directly map pixel blocks. If pure ASCII is not a requirement, consider this approach – you'll get finer detail.
- **Sixel and True Graphics in Terminal:** Some terminal emulators support **bitmap graphics protocols** (e.g., SIXEL, iTerm2, or Kitty graphics). Using these, you can display the actual image (or a very high-res approximation) in the terminal without converting to characters at all. Chafa and other tools can output in these formats [9] . The image is encoded in escape sequences. This yields by far the closest result to the original image. However, **compatibility** is limited – not all terminals support it, and those that do might have it disabled by default. If your target environment is known to support it (for instance, many Linux terminals support sixel, and Kitty has its own protocol), this is an option for the ultimate fidelity. Just be aware it's no longer "ASCII art" and more like "terminal graphics".
- **ANSI Art with Color Blocks:** Another intermediate method is to use colored half-blocks (▀ or ▄) or quarter blocks (▛ combined with background color changes) to effectively paint the image. This was a technique in BBS/ANSI art scenes. Modern usage could be printing a series of Unicode block

characters each with the right foreground/background color to represent an image. Libraries may not directly do this for you (though Chafa likely does in “Unicode mosaic” mode), but it’s a possibility for a custom solution if you need more detail than ASCII characters provide.

In summary, if ASCII art (even with color) isn’t meeting your needs for fidelity, you might explore these advanced techniques. They can deliver a nearly pixel-perfect banner in the terminal at the cost of using non-ASCII characters or special protocols. Many developers still choose ASCII/ANSI art for the nostalgia and broad compatibility – it’s universally supported and has a certain charm – but it’s good to know higher-fidelity alternatives exist if needed.

Examples of ASCII Art Banner Outputs

To wrap up, here are a couple of illustrative outputs from popular tools, demonstrating the tips discussed:

- **jp2a Example:** Using `jp2a --colors --width=80 sample.jpg` might produce something like the following (as text). It will map the image to an 80-character width ASCII art, with ANSI colors if supported. jp2a’s output quality depends on the source; a high-contrast image yields a clearly defined ASCII picture. *(Refer to jp2a’s documentation or manual for a sample output screenshot.)*
- **ascii-image-converter Example:** This tool can produce detailed color ASCII art. For instance, `ascii-image-converter logo.png -C -W 60` will output a 60-character-wide colored ASCII rendering of *logo.png*. You can add `-m "@#%*=+-. "` to force a specific character ramp, or `-c` to increase character variety. The result in a truecolor terminal will closely resemble the original logo, with the correct colors in place. The figure above with the Tux logo is generated by this tool [40], showing how color and a good character map preserve the shape and colors of the image.
- **ASCII Silhouettify Example:** For a purely monochrome, clean result, if we take a simple SVG icon and run it through ASCII Silhouettify, we’d get an output where each region of the icon is filled with a single character (or a few characters) in a solid color. The earlier embedded Ubuntu logo ASCII is an example of this approach – the tool chose appropriate characters to fill each part of the logo’s shape [46]. This works great for logos, as it ignores shading and focuses on shape.

Each tool has its own style and output, so it can be worthwhile to experiment with a few to see which gives the best results for your particular image. You might even do some manual touch-up after auto-conversion: for example, if a letter in your logo came out slightly deformed in ASCII, you can edit the text to fix that character by hand. ASCII art often benefits from artist’s tweaks even when generated by software.

Conclusion: Converting images to ASCII art banners is as much an art as a science. By picking the right tool, preparing your image wisely, and tuning character sets and colors, you can create a high-quality ASCII representation of your image. Keep in mind the environment (terminal width, font, color support) where it will be viewed for the best impact. And remember, simpler images (like vector logos) usually produce the most striking ASCII art [43] [13]. With practice and the tips above, you’ll be able to generate ASCII banners that are both technically sound and visually impressive – delivering that retro-cool effect without losing the essence of your original image.

Sources: The information and examples above draw from various resources, including tool documentation and user guides. Notable references include the jp2a manual [25] [22], discussions on AskUbuntu [47] [48], a data science workbook tutorial on ASCII art [40] [49], a Steam community guide for ASCII art tips [13] [15],

and an in-depth Medium article on ASCII art generation techniques [26] [18]. These sources provide further detail on best practices and tool capabilities for ASCII art creation.

---

[1] [2] [3] [12] [14] [16] [21] [22] [25] [29] [31] jp2a man | Linux Command Library
https://linuxcommandlibrary.com/man/jp2a

[4] [42] [47] [48] command line - How to convert graphical images to ASCII art? - Ask Ubuntu
https://askubuntu.com/questions/1210304/how-to-convert-graphical-images-to-ascii-art

[5] [6] [17] [24] [32] [36] [37] [38] [41] GitHub - TheZoraiz/ascii-image-converter: A cross-platform command-line tool to convert images into ascii art and print them on the console. Now supports braille art!
https://github.com/TheZoraiz/ascii-image-converter

[7] [11] [27] [40] [49] Viewing graphics in a terminal as the text-based ASCII art - Data Science Workbook
https://datascience.101workbook.org/07-wrangling/01-file-access/04c-view-graphic-files-terminal/

[8] [9] [45] Chafa: Terminal Graphics for the 21st Century
https://hpjansson.org/chafa/

[10] img2txt(1) — caca-utils — Debian unstable
https://manpages.debian.org/unstable/caca-utils/img2txt.1.en.html

[13] [15] [19] [20] [23] Steam Community :: Guide :: How to Make ASCII Art for the Custom Info Box on Your Steam Profile Page
https://steamcommunity.com/sharedfiles/filedetails/?id=2235568594

[18] [26] [28] [30] [44] The Art and Science of ASCII Art Generation: A Deep Dive | by Arsh Chakraborty | Dec, 2025 | Medium
https://medium.com/@chakrabortyarsh3/the-art-and-science-of-ascii-art-generation-a-deep-dive-7d04bbfe1fa9

[33] jp2a(1) — jp2a — Debian unstable - Debian Manpages
https://manpages.debian.org/unstable/jp2a/jp2a.1.en.html

[34] [35] [39] [43] [46] ASCII Silhouettify
https://meatfighter.com/ascii-silhouettify/