

Rectify — User Guide

Andy Kopra — ack@acm.org

How this guide is organized

This guide is divided into four parts:

1. **Introduction** — what Rectify is, the problem it solves, and an overview of everything it can do. Conceptual; no software or photography background assumed.
2. **Using Rectify** — a complete, task-by-task guide to the graphical application, written for someone who is not a programmer. It covers installing the supplied `Rectify.dmg` on macOS and every feature of the interface. Later sections assume some familiarity with photography and color.
3. **For programmers** — running Rectify from source (including full Linux instructions), the command-line interface, how the code is organized, and how to extend it. Assumes you have also read §2 to learn what the program does; this part is about *how it is built and driven*.
4. **Background and reference** — the design and policy decisions behind Rectify, the mathematics of every processing stage, and a self-contained color-theory primer. The earlier parts link here whenever a "why" or a deeper "how" is worth having.

Contents

1. Introduction
 - 1.1 The perspective-correction problem
 - 1.2 What you can do with Rectify
 - 1.3 How detection works, in brief
 - 1.4 A word about color
 - 1.5 Two ways to use Rectify
2. Using Rectify
 - 2.1 Installing Rectify
 - 2.2 The main window
 - 2.3 Opening images
 - 2.4 How detection works, and the vocabulary
 - 2.5 Refining the detected region
 - 2.6 The peel stack
 - 2.7 Reset and Re-open
 - 2.8 Full-image perspective correction
 - 2.9 Keystone correction
 - 2.10 Aspect-ratio correction
 - 2.11 Bow correction
 - 2.12 Color correction
 - 2.13 Saving your work
 - 2.14 Comparing before and after
 - 2.15 What Rectify remembers
 - 2.16 Keyboard shortcuts
 - 2.17 When automatic detection struggles
3. For programmers
 - 3.1 Installation

- 3.2 How the code is organized
- 3.3 The command-line interface
- 3.4 What the color modes actually do
- 3.5 Extending Rectify
- 3.6 Going deeper
- 4. Background and reference
 - 4.1 Design philosophy and policy decisions
 - 4.2 Edge detection
 - 4.3 Saturation-channel detection
 - 4.4 Contour finding and polygon approximation
 - 4.5 Corner ordering
 - 4.6 The perspective transform
 - 4.7 Aspect-ratio recovery from a single image
 - 4.8 Scoring and strategy selection
 - 4.9 Coordinate mapping through the peel stack
 - 4.10 Rectified-space nudging
 - 4.11 Detection padding
 - 4.12 The transform pipeline
 - 4.13 GUI architecture internals
 - 4.14 Color-theory primer
 - 4.15 References

1. Introduction

Rectify is a tool for straightening photographs of flat, rectangular things — paintings, museum labels, posters, tile murals, documents, signs, stamps. When you photograph a painting on a wall, you are almost never standing perfectly square to it: you shoot from a little to the side, or a little below, and the rectangular original comes out as a lopsided four-sided shape. Rectify finds that shape in your photo and warps it back into its original rectangular form.

The initial motivation for Rectify was museum photography — recovering usable, undistorted images of artwork from the angled snapshots you can actually take in a crowded gallery — but the same operation is useful any time you need the flat, square-on version of something you had to photograph at an angle.

In taking photographs of artworks, color accuracy is also important. Rectify provides methods for color correction, described in [§2.12](#).

Rectify can also remove the perspective distortion that can result when photographing buildings and other large-scale structures, restoring vertical and horizontal lines to their correct orientation in the image. This process is known as *keystone correction*, and is described in [§2.9](#).

1.1 The perspective-correction problem

A rectangle photographed at an angle appears in the image as a trapezoid, or more generally as an arbitrary quadrilateral (a four-sided shape). Recovering the original rectangle takes two steps:

1. **Detection** — finding the four corners of the object in the photograph.
2. **Transformation** — computing and applying a *perspective warp* that maps that quadrilateral back to a true rectangle.

Both steps are harder than they sound. Detection has to tell the object apart from its background, which may be a similar color, similarly bright, or busy with texture. Transformation has to undo the geometric distortion

without softening or mangling the picture. Rectify automates the first step and does the second for you, while giving you direct, hands-on control to fix anything the automation gets wrong.

1.2 What you can do with Rectify

Beyond the core "straighten this painting" operation, Rectify offers a connected set of capabilities. Each is covered step-by-step in §2; this is the map.

- **Extract a rectangular subject** from a photo — the default. Rectify detects the object's quadrilateral and warps just that region into a head-on rectangle.
- **Correct the whole image's perspective** instead of cropping — like the tilt/shift movements of a view camera. You pick something you know is rectangular (a window, a door frame), and the entire scene is corrected so that reference becomes square. See §2.8.
- **Fix keystoneing with line pairs** — when vertical or horizontal lines converge because the camera was tilted, you align a pair of lines that *should* be parallel and Rectify removes the convergence. See §2.9.
- **Peel away nested borders** — a framed painting has an outer frame, an inner mat, and the canvas. The "peel" mechanism strips these one layer at a time. See §2.6.
- **Recover the true proportions** — a rectangle shot at an angle loses its aspect ratio (the near side looks bigger). Rectify can compute the true ratio from the photo's metadata, and you can override it. See §2.10.
- **Straighten residual bowing** — a cosmetic correction for the slight inward bow that lens distortion can leave along the edges. See §2.11.
- **Correct color** — neutralize a color cast (and optionally set exposure) by sampling a known-neutral reference you placed in the shot. Three modes cover a gray card, a plain neutral patch, and a diffuse-white reference. See §2.12.

All of this is non-destructive: your source file is never altered. Rectify reads from it and produces a new, corrected output image.

1.3 How detection works, in brief

Rectify finds the object's boundary using two complementary strategies:

- **Grayscale detection** looks for boundaries where *brightness* changes — a painting against a wall of a different tone.
- **Saturation detection** looks for boundaries where *color richness* changes — a colorful tile mural against a similarly bright but grayer brick wall.

When you open an image, Rectify automatically tries both strategies across a range of sensitivity settings, scores each result for how rectangular and well-placed it is, and shows you the best one. There are no detection knobs to set in the graphical interface — the automatic sweep is always what runs, and you refine the result by hand if needed. (The command line does expose the underlying parameters for experimentation; see §3.3.) The full algorithm is described in §4.8.

1.4 A word about color

For photographing artwork, color accuracy usually matters as much as geometry. The most reliable way to get it is the same trick every raw-photo developer offers: put a known-neutral reference — a gray card, a neutral patch, a white sheet — in the frame beside the subject, then tell the software "this is neutral," and let it remove the color cast of whatever light you were shooting under.

Rectify supports three flavors of this, because "make this neutral" can mean different things: just remove the cast and leave brightness alone; remove the cast *and* set the exposure from a gray card of known reflectance; or anchor exposure to a diffuse-white reference. The operational steps are in §2.12. If the underlying ideas — color cast,

neutral references, reflectance, middle gray, coupled vs. decoupled correction — are new to you, the [color-theory primer in §4.14](#) explains them from the ground up, and the earlier sections link to it where useful.

1.5 Two ways to use Rectify

Rectify is one program with two front ends:

- **The graphical application (GUI)** is the primary way to use Rectify on every platform, and the subject of [§2](#). It is interactive: you see the detected region, drag it into place, and watch the corrected result update live. On macOS it is delivered as a ready-to-run `Rectify.dmg`; on Linux and Windows you run it from source and launch it with `python -m rectify --gui` (see [§2.1](#)). The window and every feature in it are identical on all three.
- **The command line (CLI)** drives the same engine without a window, for scripting and batch processing — correcting a whole folder of photos in a loop, for instance, without touching the GUI at all. It is available on all three platforms (on macOS from the source tree, not the app bundle) and is documented in [§3.3](#).

2. Using Rectify

This part is a complete, feature-by-feature guide to the Rectify application. It assumes no programming knowledge. The early sections assume nothing about photography either; the later ones (aspect ratio, bow, color) assume you are comfortable with ordinary photographic ideas, and point you to the [primer in §4.14](#) when a deeper concept comes up.

Everything here applies equally to macOS, Linux, and Windows — the window, the controls, and the gestures are the same on all three. Only two things differ: how you install and launch the program ([§2.1](#)), and the names of two keys.

How keys are written in this guide. Shortcuts are printed throughout with the macOS symbols, **⌘** (Command) and **⌥** (Option). **If you are on Linux or Windows, press Ctrl wherever this guide shows ⌘, and Alt wherever it shows ⌥** — so **⌘-O** means Ctrl-O, **⌘-Click** means Ctrl-Click, and **Shift-⌥** means Shift-Alt. That single substitution is the whole difference; no shortcut exists on one system and not another. Rectify labels its own tooltips and its shortcut overlay for the system you are actually on, so what you see on screen always matches your keyboard.

2.1 Installing Rectify

Rectify is delivered in two forms: as a ready-to-run application on macOS, and as source code you run with Python on Linux and Windows. Pick the section for your system; from [§2.2](#) onward the guide is the same for everyone.

macOS — the Rectify application

You have been given a file named **Rectify.dmg**. To install:

1. **Double-click Rectify.dmg**. A window opens showing the **Rectify** app icon and a shortcut to your **Applications** folder.
2. **Drag the Rectify icon onto the Applications folder** in that same window. This copies the application onto your Mac.
3. Close the window and **eject** the disk image (drag it to the Trash, or click the eject arrow next to it in a Finder sidebar). You no longer need the `.dmg`.
4. Open Rectify from **Applications** (or from Launchpad, or via Spotlight — press **⌘-Space** and type "Rectify").

The application is signed and notarized by Apple, so it opens like any other Mac app — just double-click. If macOS ever shows a caution the very first time, **right-click (or Control-click) the Rectify icon and choose "Open"**, then confirm; you only need to do this once.

Rectify runs on both Apple-silicon (M-series) and Intel Macs. There is nothing else to install — everything it needs is inside the app.

Linux — from source

There is no packaged download for Linux; you fetch the source once, install its Python dependencies into a self-contained folder, and launch the same GUI from a terminal. It is about five commands, and you only do it once. Start by cloning the public repository — no account, login, or permission is needed:

```
git clone https://git.andykopra.com/ack/rectify.git
cd rectify
```

You can also browse it in a web browser at <https://git.andykopra.com/ack/rectify/>, and download it there as a ZIP if you would rather not use git. From that point, the step-by-step instructions — including the two or three things that can go wrong on a fresh machine — are in §3.1, written to be followed without any Python knowledge.

Afterwards, each session starts like this:

```
cd ~/rectify                # wherever you put the source
source .venv/bin/activate
python -m rectify --gui      # or: python -m rectify --gui photo.jpg
```

§3.1 also shows how to reduce that to a single command you can type anywhere, or to a desktop icon you click. Once the window is open, everything from §2.2 onward applies unchanged.

Windows — from source

Windows works the same way: there is no installer, you run the program from source. Every component ships as a ready-built Windows package, so nothing has to be compiled and no system libraries are needed.

Start by cloning the public repository — no account, login, or permission is needed. In PowerShell:

```
git clone https://git.andykopra.com/ack/rectify.git
cd rectify
```

You can also browse it in a web browser at <https://git.andykopra.com/ack/rectify/> and use its download button to get a ZIP instead, which avoids installing git at all. From that point, full instructions are in §3.1.

Afterwards, each session starts like this, in PowerShell:

```
cd ~\rectify
.\.venv\Scripts\Activate.ps1
python -m rectify --gui      # or: python -m rectify --gui photo.jpg
```

2.2 The main window

When Rectify opens, the window has five areas, top to bottom:

Action bar (top). Buttons for **Open**, **Reset**, **Save**, an **Increment** checkbox, and — after a gap — a **depth** indicator with **Peel in** / **Peel out**. These are discrete actions that “do something now.” Save and its Increment mode are described in §2.13; the gap separates the file actions from the peel actions.

Image panels (center). Two side-by-side panels on a neutral gray background.

- The **left panel** always shows your original photo with the detected region drawn on top in green, with a round handle at each corner.
- The **right panel** shows the **result** — the corrected image you will save.

Both panels zoom with the mouse wheel and pan when you drag the background.

Controls (below the panels). Aspect-ratio correction, bow correction, the language selector, and font-size −/+ buttons. Detection has no controls here — it is fully automatic.

Tooltips. Hover over any control — toolbar button, checkbox, radio, slider, or label — for a short explanation in a boxed popup; the toolbar buttons also show their keyboard shortcut. Tooltips follow the selected interface language.

Action row (below the controls). **Action: Extract** / **Transform** radio buttons. **Extract** (the default) pulls the detected region out of the photo and straightens it into a rectangle. **Transform** corrects the whole image’s perspective instead, and reveals additional controls on the same row (see §2.8 and §2.9).

Status bar (bottom, italic). Shows the input filename and dimensions, the peel depth, the output dimensions, the transform mode (when active), the aspect ratio (when enabled), the undo count, and the path of the last file you saved.

Language and font size. The language dropdown switches all interface text between English, German, and Finnish. The −/+ buttons scale the font, and every margin and control scales with it — handy on very large or very small displays.

2.3 Opening images

There are several ways to load a photo:

- Click **Open** (or press ⌘-O) and choose a file.
- **Drag an image file** from your file manager (Finder, Explorer, Files, ...) directly onto the Rectify window.
- Once an image is open, press ⌘-↓ / ⌘-↑ to step to the **next** / **previous** image in the same folder (it wraps around at the ends). This is the fast way to work through a folder of photos — each one is detected fresh as it loads.

Supported formats: JPEG, PNG, TIFF, BMP, WebP, and **HEIC/HEIF** (the format iPhones save by default). HEIC photos are decoded to their standard image and converted to sRGB so colors look right everywhere; if the photo carries an HDR gain map it is ignored, which is the correct choice for documentation work. Because HEIC (like JPEG) is a *finished* picture — the camera’s white balance and tone are already applied — color correction works on the residual cast, not raw sensor data (see the [primer](#)). Saved output is always one of the standard formats; HEIC is read-only.

The **Open** dialog itself is your system’s own: Finder’s panel on macOS, Explorer’s on Windows, and your desktop’s file chooser on Linux. Its *thumbnails* therefore come from the system, not from Rectify — so on an older Linux distribution HEIC files may show a generic icon even though Rectify opens them perfectly well. A one-time fix is in §3.1.

When an image loads, Rectify immediately evaluates the detection and shows you the best result — the green quadrilateral on the left, the straightened result on the right. From there you refine by hand as needed.

2.4 How detection works, and the vocabulary

The green shape on the left panel marks the area Rectify will straighten. A few consistent names make the rest of this guide easier to follow.

- **The quad** — the four-sided green shape. It is always a quadrilateral (four straight sides), though it usually is not rectangular until Rectify straightens it.
- **Corners** — the four vertices, shown as green circular handles. Each is named for where it ends up in the result: top-left, top-right, bottom-right, bottom-left.
- **Edges** — the four sides connecting adjacent corners.

What the automatic detection does. Each time an image loads, Rectify tries both detection strategies (brightness-based and color-richness-based) across many sensitivity settings, scores each candidate for how rectangular and well-placed it is, and displays the best. You usually get a good result instantly, and fix any imprecision with the gestures in §2.5. There are no strategy or sensitivity controls in the application — the sweep is always what runs. (The full method is in §4.8.)

Two kinds of interaction: positioning vs. editing. The left panel responds differently depending on where you click inside the quad. This distinction is the key to the whole interface:

- **Editing** (click a corner handle, click near an edge, or use the arrow keys) *adjusts the detection*. The right panel updates immediately to show the effect.
- **Positioning** (click in the **center** of the quad and drag, or scroll the wheel to resize it) *moves or scales the whole quad* to select a different area. The right panel does **not** update, because the quad is now a rough selection, not a finished detection — [Peel in](#) will refine it.

In detail, the **click zones** on the left panel are:

- **On a corner handle** — the handle turns yellow; drag it, or nudge it with the arrow keys (*editing*).
- **Near an edge** — the nearest edge turns yellow; the arrow keys nudge both of its corners together (*editing*).
- **In the center** (roughly the inner half) — all four edges turn yellow. Then a **mouse drag** moves the whole quad and the **scroll wheel** grows or shrinks it (*positioning*); the **arrow keys** expand or contract it symmetrically (*editing*).
- **Outside the quad** — the panel pans (scroll-hand drag).

2.5 Refining the detected region

When the automatic detection is close but not perfect, these gestures fix it. They all update the right panel as you go.

Drag a corner. Click and drag any corner handle (it turns yellow). For accuracy, hover over the corner and **zoom in with the mouse wheel** first — the view stays anchored on the corner — then drag it precisely into place.

Nudge with the arrow keys. Click a corner (or an edge) and press the **arrow keys** to move it one pixel at a time, in the *straightened* output's sense of up/down/left/right, regardless of how the quad is tilted in the photo.

The zoom-edit-reset rhythm. A fast mouse-only loop for precise corners: hover a corner → **scroll up to zoom in** (anchored on the cursor) → drag the corner into place → **right-click to reset the zoom** → move to the next corner. The **0** key resets the zoom on both panels at once. While zoomed in, you can click **outside** the quad and drag to pan.

Preview mid-drag. While dragging a corner or edge with the left button held, **click the right mouse button** to update the right panel without letting go — preview the result, then keep adjusting.

Snap the nearest point to a click (⌘-Click). For precision, hold ⌘ and click exactly where the nearest point should sit — the closest corner jumps there, no dragging. The cursor becomes a crosshair while ⌘ is held. It is meant for use while zoomed in: zoom to the feature, then ⌘-click on it.

The alignment indicator. When two corners of an edge line up exactly — identical horizontal position (a true vertical edge) or identical vertical position (a true horizontal edge) — that edge turns **blue**. A fully axis-aligned rectangle shows all four edges blue. Use the arrow keys for pixel-perfect alignment.

Nudge or drag a whole edge. Click near an edge (not on a corner) to select it (it turns yellow), then either press the **arrow keys** (perpendicular arrows trim it in or out; parallel arrows slide it sideways) or **drag** it with the mouse. This is ideal for trimming a thin strip of leftover frame after peeling.

Undo. ⌘-Z undoes a corner adjustment; **⌘-Shift-Z** redoes it.

Selecting a different subject (center-drag). If your photo has several subjects (a gallery wall of paintings), Rectify detects the most prominent one. To grab a different one:

1. Click in the **center** of the quad (all edges turn yellow).
2. **Drag** to move the quad over the subject you want.
3. **Scroll the wheel** (with the mouse button held) to grow or shrink the quad until it roughly covers that subject.
4. Release. Then click **Peel in** — Rectify straightens the selected area and re-detects *within* it to find the precise boundary.

The quad only needs to enclose the subject with a little surrounding context; the detector finds the real edges inside it.

2.6 The peel stack

Many paintings have nested borders: an outer frame, an inner mat, then the canvas. "Peeling" strips these one layer at a time.

1. The first detection finds the outermost boundary (usually the frame edge).
2. Click **Peel in** (or press +): Rectify crops to the detected region, re-detects inside it to find the next inner boundary, and — if it finds one — moves you one layer deeper.
3. The **left panel** keeps showing your original photo; the green overlay updates to show the innermost region you have reached. The **right panel** shows the final, fully-peeled result.
4. Click **Peel out** (or press −) to back out one layer.

The **Depth** indicator shows the current level (0 = the original image). If Peel in cannot find an inner region, it tells you and disables the button until you peel out or edit the corners. Each level detects independently; the first detection prefers the *largest* good region (more context), while peel-in prefers a *tighter* inner boundary (skipping past borders).

2.7 Reset and Re-open

Two ways to restart work on an image:

Reset (⌘-D, or the Reset button) throws away the current state and starts fresh on the same pixels: back to depth 0, automatic detection from scratch, undo history cleared, aspect ratio recomputed, and back to Extract mode if you were in Transform. Use it when the current corners aren't worth keeping.

Re-open (⌘-R, keyboard only) re-reads the file from disk but *restores your edits* from Rectify's per-image memory: your corners, keystone lines, and bow value come back, and the peel stack is rebuilt. Use it when you

have edited the source photo in another program and want your tuned corners applied to the updated pixels — edit the source, save it, press ⌘-R.

	Reset	Re-open
Image	Same pixels	Re-read from disk
Corners	Re-detected from scratch	Restored from memory
Peel stack	Cleared	Rebuilt
Aspect ratio	Recomputed	Kept if you set it by hand, else recomputed
Undo history	Cleared	Cleared
Extract/Transform	Back to Extract	Preserved

2.8 Full-image perspective correction

Instead of extracting one rectangle, Rectify can correct the perspective of the *entire* image — like a view camera's tilt/shift. You choose a feature you know is truly rectangular, and the whole scene is warped so that feature becomes square.

1. In the **Action** row, select **Transform** (default is Extract).
2. A **Crop** checkbox appears. **On** (the default), the output is cropped to the largest rectangle that fits entirely inside the corrected image, removing the empty corners the warp creates. **Off**, the corrected image sits on a background of your chosen **fill color** (click the color swatch to change it).
3. The right panel updates live.

Choosing the reference. The quad now defines what "straight" means, so place its corners on something genuinely rectangular: a window, a door frame, an architectural panel — or any four points you know form a rectangle (matching column bases on opposite sides of a nave, for instance). **Manual corner placement is essential here** — drag the corners onto the reference points and fine-tune with the arrow keys.

Notes. Peeling is disabled in Transform mode. Switching from Extract to Transform preserves your Extract work and restores it when you switch back. A single correction straightens *one* plane perfectly; other planes in the scene (a second wall) may be less correct. The command line offers the same feature for batch use — see §3.3.

2.9 Keystone correction

Keystone distortion is when parallel lines converge because the camera was tilted — verticals lean together when you shoot upward at a building, horizontals when you shoot from the side. Rectify removes it from line pairs.

1. Select **Transform** in the Action row.
2. In the **Method** selector that appears, choose **Lines** (default is Quad).
3. **Vertical** and **Horizontal** checkboxes appear — Vertical is on by default. Line segments with arrow-shaped handles appear, auto-placed on the strongest edges.
4. Drag the arrow endpoints so each line lies along a feature you know is straight and parallel to the other line in its pair.

Using line pairs. **Vertical** corrects converging verticals (the common case from tilting up at a building); check **Horizontal** to also correct converging horizontals. Either or both can be on. Drag a line by its body to move it as a whole, or drag an arrow endpoint to fine-tune one end. The **Crop** and **fill color** options behave exactly as in §2.8. A keystone line turns **blue** when its endpoints share an exact horizontal or vertical position — on a real tilted building that often means the auto-placement has snapped to a stair-step artifact rather than a real edge, so nudge it onto the intended feature with ⌘-Click or the arrow handles.

Typical workflow: load a building photo with converging verticals → Transform → Lines → align the two vertical lines with two true verticals (window frames, columns) → the right panel shows them made parallel and upright → if horizontals also converge, check Horizontal and align those too. The geometry behind this is in §4.12.

Stretch. Straightening the converging lines fixes the *axes* but cannot, on its own, recover how wide the result should be relative to its height — the lines give directions, not a scale, and (unlike Transform → Quad) there's no rectangle to read a ratio from, so camera EXIF doesn't help here either. The **Stretch** slider lets you correct that residual width-to-height relationship by eye: drag until the proportions look right (squares look square, circles round). **1.00** leaves the width unchanged; above 1.00 widens, below narrows. The value shows in gray italics at 1.00 to signal it's doing nothing — exactly like Bow at 0.00. Stretch appears only in Transform → Lines and is remembered per image.

2.10 Aspect-ratio correction

A rectangle photographed at an angle comes back with the right *shape* but not necessarily the right *proportions* — the side nearer the camera looks larger, which biases the width-to-height ratio.

Automatic. When the photo carries the right metadata (its 35 mm-equivalent focal length), Rectify computes the true ratio and, if it is confident, switches the **Aspect ratio** correction on with that value. If you fix a detection by editing corners, the automatic estimate follows your corrected quad — until you set a value by hand, after which it stays put.

Manual. The **Aspect ratio** control is a label, a checkbox, and a value slider/number. Toggle the checkbox to apply it or not; set any value from 0.30 to 3.00. The ratio is width ÷ height: a square is 1.0, landscape 4:3 is 1.333, portrait 3:4 is 0.75. Use it when there is no metadata, when you know the true ratio (1.0 for a square tile, 2.35 for a CinemaScope frame), or to tune by eye.

The value's styling is a cue: **gray (italic)** means no action is needed from you — the correction is off, or a reliable value was recovered automatically from the photo's camera data (EXIF). **Black (normal)** means this image has no usable camera data (a screenshot, or EXIF stripped), so the software cannot recover the ratio on its own — drag the slider until the proportions look right. **Red** means the recovered ratio is more extreme than the slider's 0.10–10.0 range allows, so it is pinned at the limit (e.g. 10.0) and the proportions can't be fully reached — pick a less extreme reference rectangle if you need them exact. The value is remembered (not reset to 1.0), so turning the checkbox on uses what is shown. The correction preserves the longer dimension and stretches the shorter one, so no pixel detail is lost. Aspect ratio applies in **Extract mode and in Transform → Quad**; it is hidden only in Transform → Lines (which derives its own geometry from the keystone correction) and returns when you leave Lines. The math is in §4.7.

2.11 Bow correction

Even after a careful extraction, the edges of the result sometimes bow slightly at their middles — you see a sliver of frame where the painting should be, or the edge curves out past it, while the corners look right. This is leftover lens distortion, most visible when the camera was held nearly head-on.

The **Bow** slider straightens it with a corner-preserving correction: the four corners stay exactly where you put them, and the mid-edge content is moved. *Positive* values push mid-edge content outward to straighten edges that bow *inward* (**pincushion** distortion); *negative* values pull it inward to straighten edges that bow *outward* (**barrel** distortion).

The value is in **output pixels** — the maximum radial distance the correction moves content (positive = expand outward, negative = contract inward). The slider is centred on **0**, with its coloured fill growing left (negative) or right (positive) from the middle, so the two directions read symmetrically.

1. Make sure the corners are exactly on the painting's true corners (⌘-Click or drag).

2. Zoom into one bowed edge in the right panel.
3. Drag the **Bow** slider, click the up/down arrows, or type a pixel value — up for inward-bowed edges, down for outward-bowed ones. The panel updates in place without losing your zoom.
4. Stop at the smallest magnitude that straightens the edge; overshooting curves it the other way.

The slider runs from **−200 to 200 px**. Typical magnitudes are roughly **18–74 px** for iPhone main-camera photos (on a ~3000×4000 image). Because the value is an absolute pixel distance, the same setting produces the same visual amount of correction regardless of the output's size — internally it is converted to the curvature against the output's half-diagonal. Each image's bow value is remembered. This is a *cosmetic* correction anchored at the corners, not a true lens-distortion model — for mild bowing it gives a visually straight result; extreme or strongly off-center distortion won't fully straighten interior features. Extract mode only.

2.12 Color correction

When accurate color matters — and for photographing artwork it usually does — the surest way to recover it is to put a **known-neutral reference** in the shot beside the subject, then tell Rectify to neutralize the whole image to it. That removes the color cast of whatever light you were under. It is the same "click on something neutral" technique every raw developer offers, and Rectify supports three kinds of reference. If the ideas here are unfamiliar, read the [color-theory primer in §4.14](#) first.

Choosing a mode. Tick the **Color correct** checkbox and three radio buttons appear — **Gray card**, **Neutral gray**, **90% White** — with *none* selected to start, because the right choice depends on what you actually photographed:

- **Gray card** — for a photographic gray card of a known rating. It *couples* color and exposure: it removes the cast **and** sets the overall brightness from a single **Reflectance** target (default **18%**, standard middle gray; adjustable 3–80%).
- **Neutral gray** — for any patch you trust to be a neutral gray. It *decouples* color from brightness: it removes the cast but leaves the patch at the brightness the camera captured, so nothing is forced to pure white. A separate **Brightness** control (in stops, centered at 0) then raises or lowers the whole result. This is the mode for "just take the cast off."
- **90% White** — for a diffuse-white reference (a white card or clean white sheet). Like Gray card it couples color and exposure, but the target is **locked at the 90% diffuse-white standard**, so there is no target knob — just sample and go.

How to use it:

1. Tick **Color correct** and click a mode. The **swatch**, the mode's control (if any), and the **Sample size** control appear.
2. Click the **swatch**. The cursor becomes a crosshair, and a short reminder appears over the left panel telling you what to click — the next click is a *sample*, not an edit. (The reminder clears as soon as you move onto the panel.)
3. Click your reference in the **left (source) panel**. Rectify averages a small square there and corrects the result on the right. A red square marks where you sampled; re-click as often as you like to try other spots. Your source is never altered — only the result.

The controls:

- **Reflectance** (Gray card) — the tone the sampled patch is mapped to, as a reflectance percentage (the number printed on a gray card). Default **18%** (middle gray). Set it to your card's actual rating, or use it as an exposure lever — higher is brighter. Reflectance is *linear*: 18% is middle gray and 50% is already quite bright, so don't reach for 50% expecting "neutral." Range 3–80%.
- **Brightness** (Neutral gray) — an overall lightness adjustment in stops, centered at **0** (no change). At 0, only color is corrected and the captured brightness is kept; go negative if neutralizing the cast pushes a

bright area to clip, positive to lighten. It never affects the color balance, only the level.

- **90% White** has no target control — the target is the fixed 90% standard.
- **Sample size** — the side length, in source pixels, of the averaged square. **1** samples the single clicked pixel; **10** (default) averages a 10×10 block. Increase it for a larger, more representative patch; decrease it when the neutral area is tiny.
- **The swatch** shows the color you sampled, so you can see the cast you grabbed. It turns **red** when the spot is unusable — clipped to white or crushed to black, where the correction can't be computed. (White references sit near the top of the range, so pick a sheet that isn't blown out.)
- **The status bar** shows the sampled color while correction is active (the mode name and the three channel values, normalized 0–1). For a truly neutral reference the three numbers are close; if blue reads much higher than red, the spot is bluish and neutralizing it will warm the whole image — a sign the spot wasn't really neutral.

Working zoomed in. Picking a reference, switching modes, and adjusting Reflectance, Brightness, or Sample size all keep the right panel's current zoom and pan, so you can judge the correction on a detail. The sliders are *debounced* — they update the value immediately but wait until you pause to recompute the (slow) image — and saving always reflects the latest values.

Where to place the reference. Anywhere in the frame outside the subject is fine; sample it on the left panel where the whole photo is visible. Rectify reads the correction from the source but applies it to the extracted or transformed *output*, so the reference need not survive into the final crop. Color correction works in every mode (Extract, Transform → Quad, Transform → Lines).

Reusing one reference across a batch. The measured reference is remembered, so if you shoot a series under the same light you only need the reference in *one* frame:

- A new image opens with **Color correct off**. Turn it on, and if you haven't sampled on *that* image, Rectify borrows a **snapshot** of your last measured reference. The swatch shows the borrowed color and there is **no red marker** — the marker's absence tells you the correction is borrowed.
- The swatch's two clicks differ: **left-click** arms a new pick (then click a neutral area); **right-click** re-applies your last measured reference to the current image (use it to push an updated reference onto an image that already had one).
- It is a **snapshot, not a live link** — borrowing copies the value; re-picking the reference frame later won't change images that already borrowed the old one.
- A reference is only valid under the **same light** as the frame you measured it on. Move to different lighting and pick a fresh one.

What each image remembers. Every image keeps its own on/off state, **mode**, sample point, and any borrowed color, so stepping between images with ⌘-↑/↓ preserves exactly how each is corrected. A newly opened image starts with **no mode chosen**. The three modes keep **separate** references on the same image, so you can sample a gray card and a white sheet independently and switch between them without losing either. **Reset (⌘-D)** clears the current image's correction but keeps the remembered references, so a batch calibration in progress survives.

An honest limitation. This corrects the *light's color* faithfully to the file's encoding, but it cannot undo what the camera already baked into a JPEG or HEIC, nor fix *uneven* lighting — a single global correction assumes the light's color is the same across the whole frame. For a painting lit evenly by one source it works very well; for mixed lighting it corrects the cast on average.

2.13 Saving your work

A single **Save** button writes the corrected result, with two modes set by the **Increment** checkbox beside it:

The default output name is the source image's name with **_rectified** appended — for example `hotel.png` becomes `hotel_rectified.png` — so a save never overwrites the original.

- **Increment off** (default) — Save opens a dialog, pre-filled with that default name. The extension you give picks the format (png, jpg, jpeg, tiff, tif, webp, or bmp). An unsupported extension is rejected with a message listing the supported ones. Type no extension and the last type you used is added (PNG to begin with).
- **Increment on** — Save writes the next auto-numbered file with one click and no dialog: *name_rectified_1.ext*, then *name_rectified_2*, ... Rectify scans the folder for the highest existing number and uses the next free one, so you never overwrite an earlier save.

Both modes share the **last folder you saved to** (the source image's folder the first time) and the **last file type**. The folder, type, and Increment setting are remembered across sessions. **⌘-S** also saves, using whichever mode is active. The status bar shows the full path of the last file you saved.

2.14 Comparing before and after

Hold the **Space bar** to temporarily show the *original* image (without the green overlay) in the right panel; release it to return to the corrected result. A quick way to judge the correction.

2.15 What Rectify remembers

Rectify saves your preferences when you close it and restores them next time, so you can pick up exactly where you left off — same image, same corners, same settings.

Saved:

Setting	Examples
Aspect ratio	On/off and value (auto per image, or your per-image override)
Extract/Transform	Mode, Crop toggle, fill color
Saving	Last folder, last file type, Increment on/off
Interface	Language, font size, window size/position, panel split
Per-image memory	For every image you've touched: corner positions at all peel depths, keystone line pairs, bow value, color-correction state, and any manual aspect override
Last image	The file to reopen on next launch

Not saved: undo/redo history, and detection settings (there are none to remember — the automatic sweep runs on every load).

Because each image's edits live in this per-image memory, you can prepare a whole set in one session — adjusting corners, lines, bow, color, and aspect on each — just by stepping between them with **⌘-↑/↓**, without saving to disk between switches.

The preferences file is stored per-user, so several people sharing one computer keep independent settings:

Platform	Location
macOS	~/Library/Application Support/<username>/Rectify/settings.json
Linux	~/.local/share/<username>/Rectify/settings.json
Windows	%APPDATA%\<username>\Rectify\settings.json

(It is plain JSON and can be edited by hand if you ever need to, though normally you never will. Deleting it resets Rectify to a first-run state.)

2.16 Keyboard shortcuts

Shortcut	Action
⌘-O	Open image
⌘-S	Save result (PNG if no extension given)
⌘-D	Reset (clear state, re-detect from scratch)
⌘-R	Re-open current file from disk (restores your cached edits)
⌘-↓ / ⌘-↑	Next / previous image in the folder (wraps)
⌘-Z / ⌘-Shift-Z	Undo / redo a corner adjustment
+ or =	Peel in
–	Peel out
Space (hold)	Show the original in the right panel (before/after)
Arrow keys	Nudge the selected corner, edge, or whole quad
Mouse wheel	Zoom (or adjust the element under the cursor with Shift held)
Shift-wheel	Adjust the quad element under the cursor
⌘-Left-click	Snap the nearest point to the click position
⌘-click ×2	Draw an alternate-line annotation between two clicks
Right-click (or ^-click)	Reset zoom on the clicked panel
0	Reset zoom on both panels
Esc	Clear temporary marks
Shift-⌘ (hold)	Show the keyboard-shortcut overlay (release to dismiss)

On Linux and Windows: read **Ctrl** for ⌘ and **Alt** for ⌘ throughout the table — ⌘-O is Ctrl-O, ⌘-click is Ctrl-click, Shift-⌘ is Shift-Alt. Everything else is identical.

Hold **Shift-⌘** at any time to see this table without leaving the window, with the key names for the system you are on. The same table is printed by `rectify -k` from the command line.

Two macOS-only details are worth knowing. **⌘-click** snaps the nearest point, while **^-click** — the *physical* Control key — is simply how a Mac delivers a right-click on a one-button mouse or trackpad, and so resets the panel's zoom. And where a combination includes Shift, the on-screen overlay follows Apple's ordering and puts it first, as **↑-⌘-Z** and **⌘-↑** — the same keys as the **⌘-Shift-Z** and **Shift-⌘** written above, listed in the other order.

2.17 When automatic detection struggles

When detection grabs the wrong corners or misses the subject, your tools in the application are the manual gestures from §2.5: drag corners and edges, snap the nearest point with ⌘-Click, nudge with the arrow keys, or use the center-drag "select a subregion" workflow to point detection at a different part of the image. Common situations:

- **Low contrast between subject and background** — edit the corners by hand; often the boundary is clear to your eye even when the detector hesitates.

- **Busy or textured backgrounds (brick, patterned wallpaper)** — detection usually still finds the subject; correct any stray corner by dragging.
- **Several rectangles in the frame** — Rectify picks the most prominent; use center-drag to move the quad over the one you want, then Peel in.
- **Frames within frames** — use the [peel stack](#), peeling in past each layer.
- **Black-bordered screenshots** — handled automatically; the black border is detected and the content boundary found without intervention.
- **Very small or thin regions** — if peeling would produce a degenerate region, Rectify shows a message instead of failing.

If you want to push detection harder by hand — choosing a strategy, raising the sensitivity, changing the edge thresholds — those controls exist only on the command line; see [§3.3](#).

3. For programmers

This part covers running Rectify from source, driving it from the command line, how the code is laid out, and how to extend it. It assumes you have read [§2](#) to learn what the features *do* — here we focus on how the program is built and operated, and it links to [§4](#) for the algorithms behind each stage.

3.1 Installation

macOS (the app)

For day-to-day use on a Mac, install the supplied `Rectify.dmg` exactly as in [§2.1](#) — drag the app to Applications. That bundle is a self-contained build (Python, Qt, and OpenCV included) and needs nothing else. To *develop* on macOS instead, follow the from-source steps below; they work on macOS with Homebrew or python.org Python (Apple silicon and Intel).

Linux (from source)

Linux is the platform Rectify is developed on, so it is the best-tested of the three — but there is no packaged download the way macOS has its `.dmg`. You run it from source, and the **GUI is the primary interface here as everywhere else** (the CLI in [§3.3](#) is the same engine without a window, for scripting and batch work).

Prerequisites: Python 3.10 or later, pip, git, and a graphical desktop. Development happens on Ubuntu with GNOME; any current distribution with those pieces works, and nothing below is Ubuntu-specific except the apt command lines, which have direct equivalents in dnf, pacman, and zypper.

1. **Clone the public repository.** It is open to everyone — no account, login, or permission is required, and it can be browsed (and downloaded as a ZIP) at <https://git.andykopra.com/ack/rectify/>.

```
git clone https://git.andykopra.com/ack/rectify.git
cd rectify
```

2. **Make sure Python can create virtual environments.** Debian and Ubuntu ship Python without this piece, and step 3 fails with `ensurepip is not available` if it is missing:

```
sudo apt install python3-venv python3-pip
```

3. **Create a virtual environment and install dependencies:**

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

This pulls in OpenCV, NumPy, PySide6 (Qt), Pillow, and pillow-heif (HEIC/HEIF reading — it bundles its own libheif, so no system library is required). Everything lands inside `.venv/`; nothing is installed system-wide, and deleting the source folder removes all of it.

4. **Qt system libraries.** If PySide6 fails to start, install the X11/XCB libraries Qt needs. On Ubuntu/Debian:

```
sudo apt install libxcb-xinerama0 libxcb-cursor0
```

5. Run it:

```
source .venv/bin/activate          # if not already active

python -m rectify --gui            # GUI, open a file later
python -m rectify --gui photo.jpg # GUI with an image
python -m rectify photo.jpg -o rectified.jpg # CLI, no window
```

6. **(Optional) HEIC thumbnails in the file chooser.** Rectify reads HEIC regardless, but the **Open** dialog's thumbnails come from your desktop, which on older distributions can't render iPhone HDR HEICs. Run `scripts/install_heic_thumbnailer.sh` once to enable them (it self-skips if your system already handles HEIC).

Launching without the two-step dance. The virtual environment's own interpreter can run Rectify directly, so a one-line wrapper lets you start it from any directory, with relative filenames intact. Save this as `~/bin/rectify` and `chmod +x` it, substituting your own path:

```
#!/bin/sh
exec env PYTHONPATH="$HOME/rectify" "$HOME/rectify/.venv/bin/python" \
    -m rectify "$@"
```

Then `rectify --gui photo.jpg` works anywhere. For a clickable icon, point a desktop entry at that wrapper — save the following as `~/.local/share/applications/rectify.desktop`:

```
[Desktop Entry]
Type=Application
Name=Rectify
Exec=/home/YOUR_USER/bin/rectify --gui %f
Terminal=false
Categories=Graphics;Photography;
MimeType=image/jpeg;image/png;image/tiff;image/webp;image/bmp;image/heif;
```

It then appears in your application menu, and images can be opened with it from the file manager.

If the window misbehaves on Wayland. Qt picks Wayland automatically in a Wayland session. If Rectify fails to start, or the window's sizing, cursors, or tooltip placement look wrong there, force the X11 path for one run:

```
QT_QPA_PLATFORM=xcb python -m rectify --gui
```

If that fixes it, put `export QT_QPA_PLATFORM=xcb` in the wrapper script above. This is worth trying first for any display-related oddity, particularly on NVIDIA drivers, where Wayland compositor bugs are still common.

On Linux the per-user settings file lives at `~/.local/share/<username>/Rectify/settings.json`; all three platforms' paths are listed in §2.15.

Windows (from source)

On Windows, Rectify is run from source. Every dependency publishes a Windows wheel — PySide6 bundles Qt and pillow-heif bundles libheif — so `pip install` is the entire build step: no compiler is needed, and no system libraries have to be installed the way they do on Linux. The **Open** dialog is the standard Explorer dialog, complete with image thumbnails.

Prerequisites: Python 3.10–3.13 from python.org, installed with "Add python.exe to PATH" ticked. git is optional — the repository can be downloaded as a ZIP instead.

1. **Clone the public repository and create a virtual environment.** The repository is open to everyone — no account, login, or permission is required, and it can be browsed (and downloaded as a ZIP, if you would rather not install git) at <https://git.andykopra.com/ack/rectify/>. In PowerShell:

```
git clone https://git.andykopra.com/ack/rectify.git
cd rectify
py -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
```

From Command Prompt the only difference is the activation line, `.venv\Scripts\activate.bat`. If PowerShell blocks the activation script, permit it for that window alone with `Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`.

2. Run it:

```
python -m rectify --gui           # GUI, open a file later
python -m rectify --gui photo.jpg # GUI with an image
python -m rectify photo.jpg -o rectified.jpg # CLI, no window
```

On Windows the per-user settings file lives at `%APPDATA%\<username>\Rectify\settings.json`.

Two failure modes worth naming. If PySide6 raises `ImportError: DLL load failed while importing QtCore`, install the Microsoft Visual C++ 2015–2022 Redistributable (x64), which Qt links against; it is already present on most Windows machines. If `py` is not recognized, the Python launcher was not installed — use `python -m venv .venv` instead.

Platform maturity. Windows is not yet part of the regular test rotation, and the code carries no Windows-specific branches (the only per-platform code in the tree is a handful of macOS conditionals, which fall through to the generic path). The areas most likely to reveal a problem, and therefore worth checking deliberately on a first run, are: opening an iPhone HEIC/HEIF photo; display scaling at 125% and 150%, against Rectify's own font scaling; the Ctrl-based keyboard shortcuts; saving to a path containing spaces; and building `dist\rectify.exe` with `pyinstaller rectify.spec`.

3.2 How the code is organized

rectify/	
├── __main__.py	Entry point (python -m rectify)
├── cli.py	Argument parsing, CLI pipeline
├── gui.py	PySide6 GUI (QMainWindow, QGraphicsView panels)
├── detect.py	Detection engine (strategies, scoring, evaluation)
├── transform.py	Perspective transform, color correction (NumPy only)
└── utils.py	Image I/O, OpenCV*Qt conversion

Dependencies flow one way: `cli.py` and `gui.py` drive `detect.py` and `transform.py`, which depend only on `utils.py`. Crucially, **`detect.py` and `transform.py` have no GUI dependencies** — they operate on NumPy arrays alone, so the detection and transform engines can be reused in other contexts (a web API, a mobile app, a batch processor) without modification.

The algorithm internals behind these modules — edge detection, scoring, the transform math, aspect recovery, the peel-stack coordinate mapping — live in §4. The GUI's internal structure (class hierarchy, the QGraphicsView coordinate system, undo, cursor logic, settings persistence) is in §4.13.

3.3 The command-line interface

The CLI drives the same engine as the GUI without a window — for scripting and batch processing, and as the primary interaction mode on Linux. Each option below links to the GUI description of the same capability.

Basic extraction (auto strategy and sensitivity — the same automatic sweep the GUI runs, see §2.4):

```
python -m rectify photo.jpg -o rectified.jpg
```

Detection parameters (the knobs the GUI deliberately hides — see §2.17). `--strategy` chooses the detection channel; `-s/--sensitivity` and the Canny/area/epsilon parameters tune it directly:

```
python -m rectify photo.jpg -o rectified.jpg --strategy saturation -s 0.7
python -m rectify photo.jpg -o rectified.jpg --blur 3 --canny-low 30 \
    --canny-high 100
```

`--strategy {auto,grayscale,saturation}`, `-s/--sensitivity 0.0-1.0`, `--blur`, `--canny-low`, `--canny-high`, `--min-area`, `--epsilon`. The sensitivity-to- parameters mapping is in §4.2.

Peeling (the [peel stack](#)):

```
python -m rectify photo.jpg -o rectified.jpg --peel 1      # one layer
python -m rectify photo.jpg -o rectified.jpg --remove-frame # = --peel 1
python -m rectify photo.jpg -o rectified.jpg --peel 3      # three layers
```

Full-image perspective correction (the [Transform mode](#)):

```
python -m rectify photo.jpg -o corrected.jpg --full-image
python -m rectify photo.jpg -o corrected.jpg --full-image --full-image-crop
python -m rectify photo.jpg -o corrected.jpg --full-image --fill-color "#808080"
```

Incremental, auto-numbered output (the same scheme as the GUI's [Save](#)). With no `-o`, files are written as *prefix_N.ext*:

```
python -m rectify photo.jpg \
# ./rectify_1.png, ...
python -m rectify photo.jpg --dir output/ --prefix museum --ext jpg
```

The sequence number *N* comes from scanning the output directory for existing files matching the prefix across *all* extensions, so `museum_3.png` and `museum_5.jpg` both count and the next is 6.

Batch processing (Linux/macOS shell):

```
for img in photos/*.jpg; do
    python -m rectify "$img" --dir rectified/ --prefix painting --peel 1
done
```

Debug logging records the detection pipeline's decisions (contour counts, area filters, candidate scoring, the two-pass sweep, final selection), each entry timestamped — useful for diagnosing why a detection was chosen or missed:

```
python -m rectify photo.jpg -o out.jpg --debug \
# → rectify_debug.log
python -m rectify --gui photo.jpg --debug my_debug.log \
# custom path, GUI too
```

Keyboard reference: `rectify -k` (also `--keyboard`) prints the shortcut table (the same one the GUI shows on Shift-⌘), with the key names of whichever system it is run on.

Note: color correction ([§2.12](#)) is interactive — it depends on clicking a reference in the image — so it is a GUI-only feature; the CLI does not apply color correction.

3.4 What the color modes actually do

All three modes ([§2.12](#)) share the same shape: sample a small neutral patch, compute one **per-channel linear-light gain**, and multiply the output image by it. They differ only in how the gain is derived. The theory — reflectance, middle gray, why the math is done in linear light — is in the [primer, §4.14](#). The implementation lives in `transform.py`.

Sampling. `sample_patch_bgr(image, cx, cy, radius)` averages a `radius × radius` square (`radius 1` = the single clicked pixel) and returns the mean BGR as float in 0–255. A patch is rejected (the swatch goes red) if any channel is at or beyond the clip guards (≤ 3 crushed, ≥ 252 clipped), where a per-channel gain would blow up or divide by $\sim$ zero.

Coupled correction — Gray card and 90% White — `gray_correction_gains(sampled, target_reflectance)`:

```
gain_c = target_reflectance / sampled_linear_c      (von Kries, per channel)
```

The sampled BGR is converted from sRGB to linear light; the gain maps it to `target_reflectance` (a linear quantity, used directly). Because the target is the *same* for all three channels, one gain set simultaneously removes the cast (equal targets $\Rightarrow$ neutral) **and** sets exposure (the target level). **Gray card** passes your Reflectance

percentage as the target (default 0.18); **90% White** passes a fixed 0.90. That single difference — and the fact that the Gray card slider caps at 0.80 — is the entire distinction between the two modes.

Decoupled correction — Neutral gray — `white_correction_gains(sampled, brightness_stops)`:

```
gain_c = 2*brightness_stops * (L_patch / sampled_linear_c)
```

where `L_patch` is the patch's Rec. 709 linear luminance. At `brightness_stops = 0` the sampled patch comes out neutral ($R = G = B$) at *exactly its original luminance* — the cast is removed but nothing is forced to white, so a specular glint or a visible bulb keeps its headroom. The brightness factor then scales the whole result. This is why the mode is "just remove the cast."

Applying the gain. `apply_gray_correction(image, gains)` multiplies the BGR image by the per-channel gains in linear light and re-encodes to sRGB. The same gain is applied to whatever the right panel shows (extracted or transformed), so the reference need not survive into the final crop.

Persistence keys. Internally the modes are "gray" (Gray card), "white" (Neutral gray — the original decoupled mode; the key is kept for backward compatibility), and "white90" (90% White). Each keeps its own sampled point, color, radius, and sticky value, so the three references coexist on one image.

3.5 Extending Rectify

Add a detection strategy. In `detect.py`, write a function in the mold of `_detect_grayscale / _detect_saturation` (prepare a channel, call `_multipass_canny` with suitable thresholds, return ordered corners or None); add its name to the STRATEGIES tuple; add a case in `detect_quad()`; add it to the candidate loop in `evaluate_strategies()`; and add a `--strategy <name>` choice in `cli.py`. (The GUI exposes no strategy selector; the CLI does.)

Add scoring criteria. Modify `_quality_score()` or `_score_quad()` in `detect.py`. They take corners plus image dimensions and return a float (higher is better); keep it roughly normalized to 0.0–1.0 so the thresholds in `evaluate_strategies()` still hold. See §4.8 for what the existing terms mean.

Add GUI controls. Controls are built in `_build_ui()` in `RectifyMainWindow`. Connect them to detection or display methods via Qt signals, and wrap programmatic updates in `blockSignals(True/False)` to avoid cascading recomputation. See §4.13.

Diagnostic aids built into the GUI. `⌘-E` toggles a Canny edge-detection overlay on the left panel (red edges over the source, using the grayscale defaults) — useful for seeing what the detector sees. `⌘-Delete` (or `⌘-Backspace`) clears the settings cache and resets all controls to defaults. (As everywhere in this guide, read those as `Ctrl-Shift-E` and `Ctrl-Shift-Delete` on Linux and Windows.) Two environment variables draw layout-debugging overlays for GUI work: `RECTIFY_DEBUG_BORDERS=1` outlines every widget, and `RECTIFY_DEBUG_BASELINES=1` draws a red line at each text widget's baseline.

3.6 Going deeper

§4 covers the design decisions and the mathematics of every stage, plus the color-theory primer. Before modifying a subsystem, read the reasoning behind it:

Subsystem	Where it is explained
Detection — strategies, scoring, padding	§4.2–§4.4, §4.8, §4.11
Perspective transform and keystone correction	§4.5–§4.6, §4.12
Geometry corrections — aspect ratio, bow	§4.7, §4.12

Subsystem	Where it is explained
Editing gestures and the peel stack	§4.9–§4.10
Color correction and input formats	§4.14
GUI architecture, state, and persistence	§4.13

4. Background and reference

This part explains *why* Rectify behaves as it does and *how* each stage works mathematically, and ends with a self-contained color-theory primer. It is meant to be dipped into — the earlier parts link here by section number.

4.1 Design philosophy and policy decisions

A handful of deliberate choices shape the whole program.

Show results, not knobs. The graphical interface exposes *no* detection parameters — no strategy, sensitivity, threshold, or area controls. The reasoning is that a person looking at a photo can see instantly whether the corners are right and can drag them if not, whereas a panel of sliders invites fiddling without insight. So detection runs a full automatic sweep on every load ([§4.8](#)), and *all* refinement is direct manipulation of the result. The underlying parameters still exist for experimentation, but only on the command line ([§3.3](#)).

Positioning vs. editing. The single most important interaction rule ([§2.4](#)) is that some gestures *edit* the detection (and update the result live) while others *position* a rough selection (and intentionally do not update the result). A center-drag or wheel-*resize* is a coarse "look over here" that [Peel in](#) will refine; updating the result from it would imply a precision the selection doesn't have.

Prefer larger first, smaller when peeling. Initial detection prefers the *largest* high-quality region (capture the most context); peel-in prefers the *smallest* (find the meaningful inner boundary, skipping past frame borders rather than trimming slivers). Both go through the same selection logic, which first prunes to candidates within a tight quality tolerance and only then breaks ties by area — so a slightly-larger but visibly-worse quad cannot win on area alone. A peel "area floor" additionally blocks high-quality outliers whose edge has locked onto a feature far inside the parent. Details in [§4.8](#).

Each peel layer in its own coordinate space. The peel stack does not map coordinates between layers; each layer is simply the rectified image of the one above it, detected afresh ([§4.9](#)). This is simpler and more robust than the alternative of mapping every detection back to original coordinates, and it means the detector always sees exactly what you see.

Pad only the original. Detection pads the image edges outward so tightly-framed subjects get a margin for the edge detector and the margin score ([§4.11](#)). Padding is applied only to the original image; on a rectified peel result, replicating edge pixels would invent false boundaries.

HEIC is treated as documentation source. HEIC/HEIF photos are decoded to their standard-dynamic-range base image and converted from Display P3 to sRGB; any HDR gain map is ignored. For print/catalog documentation that is the correct choice — you want a predictable SDR image in a known color space, not an HDR rendering.

Aspect ratio: automatic, overridable, and lossless. When metadata permits, the true ratio is computed and applied ([§4.7](#)); editing corners updates the estimate until you set a value by hand, after which your value sticks. The correction preserves the longer output dimension and stretches the shorter one, so it never discards pixels.

Bow is cosmetic by design. The bow correction ([§2.11](#)) is a corner-anchored radial nudge, not a calibrated lens-distortion model. It is meant to clean up the few tenths of a percent of residual pincushion (edges bow inward)

or barrel (edges bow outward) distortion left after a camera's own correction — deliberately simple, and honest about not being a true undistort.

Color correction corrects the light, faithfully but globally. The color tools (§2.12, §3.4) remove the illuminant's cast as encoded in the file. They cannot undo what a camera baked into a JPEG/HEIC, and they assume the light's color is uniform across the frame (a single global gain). The three modes exist because "make this neutral" has two honest meanings — *also set exposure* (coupled) or *leave exposure alone* (decoupled) — plus a dedicated diffuse-white anchor; see the [primer, §4.14](#). The measured reference is a *snapshot* when reused across a batch, never a live link, so re-measuring one frame never silently changes images that already borrowed the old value.

Nothing is destructive. The source file is never modified. All edits live in a per-image cache so a whole set can be prepared in one session and saved on demand.

4.2 Edge detection

Rectify finds boundary pixels with the Canny edge detector (Canny, 1986), which operates in four stages:

1. **Gaussian blur** — smooths the image to reduce noise; the kernel size comes from the sensitivity parameter. More blur suppresses noise but can soften real edges.
2. **Gradient computation** — the intensity gradient magnitude and direction at each pixel, via Sobel operators:

$$\begin{aligned} G_x &= \partial I / \partial x, & G_y &= \partial I / \partial y \\ \text{magnitude} &= \sqrt{G_x^2 + G_y^2} \\ \text{direction} &= \text{atan2}(G_y, G_x) \end{aligned}$$

3. **Non-maximum suppression** — thins edges to single-pixel width by keeping only local maxima along the gradient direction.
4. **Hysteresis thresholding** — two thresholds: pixels above high are strong edges; pixels between low and high are kept only if connected to a strong edge; pixels below low are discarded.

The two thresholds are the primary sensitivity controls. Rectify maps the user-facing sensitivity (0.0–1.0) to them, and to three more parameters:

Parameter	s = 0.0	s = 1.0	Effect of increase
Blur kernel	7	3	Less smoothing, preserves detail
Canny low	70	15	Detects weaker edges
Canny high	180	80	Detects weaker edges
Min area ratio	0.05	0.01	Accepts smaller regions
Epsilon ratio	0.04	0.015	Tighter polygon approximation

The threshold lines explicitly:

$$\begin{aligned} \text{canny_low} &= 70 - 55 \times \text{sensitivity} & (70 \rightarrow 15) \\ \text{canny_high} &= 180 - 100 \times \text{sensitivity} & (180 \rightarrow 80) \end{aligned}$$

After edge detection, morphological operations (dilation, closing) bridge small gaps. Rectify uses a multi-pass strategy: if the first Canny pass yields no quadrilateral, it tries morphological closing (7×7 kernel), then

progressively lower thresholds (60% and 40% of the original).

Reference: J. Canny, "A computational approach to edge detection," *IEEE TPAMI*, vol. 8, no. 6, pp. 679–698, 1986.

4.3 Saturation-channel detection

When subject and background are similarly bright but differ in color richness (a ceramic mural on a brick wall), the luminance channel gives poor edges. The saturation strategy instead works on the **S** channel of HSV:

```
H = hue (color angle, 0°–360°)
S = saturation (color purity, 0–255)
V = value (brightness, 0–255)
```

Two findings from development shape it: **no Gaussian blur** is applied (saturation transitions are inherently smoother than luminance ones, and blurring smears away the subtle edges that matter), and **fixed Canny thresholds** (low = 20, high = 60) are used, tuned for the saturation channel's distribution.

4.4 Contour finding and polygon approximation

OpenCV's `findContours` traces the boundaries of connected edge regions. Rectify filters them by area (rejecting those below a minimum fraction of the image or above 95% of it) and sorts by area, descending.

Each candidate is simplified with the Douglas–Peucker algorithm (Ramer, 1972; Douglas & Peucker, 1973) via `approxPolyDP`, which removes points within a tolerance ϵ of the simplified line:

```
 $\epsilon$  = epsilon_ratio × perimeter
```

with `epsilon_ratio` typically 0.02–0.04. A simplified polygon with exactly four vertices is accepted as a quadrilateral candidate. If none is found, Rectify falls back to the convex hull of the largest contour, approximated to four vertices.

References: U. Ramer, *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244–256, 1972; D. H. Douglas and T. K. Peucker, *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.

4.5 Corner ordering

Four unordered points are assigned to [top-left, top-right, bottom-right, bottom-left] by a sum/difference heuristic:

```
For each point (x, y):  sum = x + y,  diff = y - x

top-left      = minimum sum      (closest to origin)
bottom-right  = maximum sum      (farthest from origin)
top-right     = minimum diff     (far right, near top)
bottom-left   = maximum diff     (far left, near bottom)
```

For a roughly upright rectangle the top-left corner minimizes both x and y (hence minimum $x+y$) while the bottom-right maximizes both.

4.6 The perspective transform

A perspective (projective) transform maps a quadrilateral to a rectangle with a 3×3 homography $\mathbf{H}$. For four source points $\mathbf{p}_i$ and destinations $\mathbf{q}_i$:

$$\lambda_i [\mathbf{q}_i; 1] = \mathbf{H} [\mathbf{p}_i; 1]$$

with scalar factors λ_i . The system has 8 degrees of freedom (the 9 entries of $\mathbf{H}$ minus scale), exactly determined by 4 correspondences. OpenCV's `getPerspectiveTransform` solves it and `warpPerspective` remaps every pixel with bilinear interpolation. The destination rectangle is sized as:

$$\begin{aligned} \text{width} &= \max(\text{dist}(\text{TL}, \text{TR}), \text{dist}(\text{BL}, \text{BR})) \\ \text{height} &= \max(\text{dist}(\text{TL}, \text{BL}), \text{dist}(\text{TR}, \text{BR})) \end{aligned}$$

The maximum of each opposite pair is used because the side nearer the camera is less foreshortened and so a better estimate of the true dimension.

Reference: R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004 (ch. 2 and 4).

4.7 Aspect-ratio recovery from a single image

A rectangle photographed at an angle loses its true proportions — the near side appears larger. Rectify recovers the ratio by homography decomposition with the camera intrinsics, following Zhang (2000) and Criminisi et al. (2000).

Camera intrinsic matrix. A pinhole with focal length f (pixels) and principal point at the image center:

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

The focal length in pixels comes from EXIF:

$$f_{\text{pixels}} = f_{35\text{mm}} \times \text{image_width} / 36$$

where $f_{35\text{mm}}$ is `FocalLengthIn35mmFilm` and 36 mm is the 35 mm frame width.

Homography decomposition. The homography mapping a unit square to the image quad factors as $\mathbf{H} = \mathbf{K} [\mathbf{r}_1 \mathbf{r}_2 \mathbf{t}]$, so $\mathbf{M} = \mathbf{K}^{-1}\mathbf{H} = [\mathbf{r}_1 \mathbf{r}_2 \mathbf{t}]$. For a unit-square input, $\mathbf{r}_1$ and $\mathbf{r}_2$ are the width and height directions in camera space. Since rotation columns are orthonormal, a square requires $|\mathbf{r}_1| = |\mathbf{r}_2|$; for a rectangle of aspect $a = W/H$ the columns scale differently, giving:

$$a = W/H = |\mathbf{r}_1| / |\mathbf{r}_2|$$

Orthogonality check. The decomposition assumes no lens distortion. To catch cases where that fails (common with phone cameras), Rectify checks

$$\cos \theta = |(\mathbf{r}_1 \cdot \mathbf{r}_2) / (|\mathbf{r}_1| |\mathbf{r}_2|)|$$

and treats the estimate as unreliable (not applied automatically) when $\cos \theta > 0.05$. The user can still set the ratio by hand.

Quality preservation. Applying the correction preserves the longer output dimension and stretches the shorter one — upscaling the dimension with less information rather than discarding pixels.

References: Z. Zhang, *IEEE TPAMI*, vol. 22, no. 11, pp. 1330–1334, 2000; A. Criminisi, I. Reid, A. Zisserman, *IJCV*, vol. 40, no. 2, pp. 123–148, 2000.

4.8 Scoring and strategy selection

Geometric quality score. `_quality_score` rates a quad without regard to area (so large detections aren't penalized):

$$\text{quality} = 0.35 \times \text{angle_score} + 0.35 \times \text{margin_score} + 0.30 \times \text{persp_score}$$

(With `USE_PERSPECTIVE_SCORE = False`: $0.5 \times \text{angle} + 0.5 \times \text{margin}$.)

- **Angle score** — rectangularity: $\max(0, 1 - \text{avg_deviation} / 30^\circ)$, where `avg_deviation` is the mean absolute difference of each interior angle from 90° .
- **Margin score** — distance from the image edges: 0 if any corner is within 2 pixels of an edge (it has grabbed the boundary rather than the subject), else $\min(\text{min_margin} / (0.05 \times \text{min_dim}), 1.0)$.
- **Perspective plausibility** — $\max(0, 1 - |\cos(\text{angle})| / 0.3)$, where the cosine is between the two rotation-matrix columns recovered from homography decomposition (assuming a 50 mm focal length). A true projected rectangle gives orthogonal columns ($\cos = 0$, score 1.0); an implausible quad gives $|\cos| \geq 0.3$ (score 0.0). This rejects accidental 4-vertex contours with good angles and margin that no perspective could have produced.

A separate full composite score, `_score_quad`, additionally weights area (ideal 15–70% of the image) and is used where absolute quality matters.

The two-pass sweep. `evaluate_strategies` runs a coarse pass over both strategies at sensitivities 0.0–1.0 in steps of 0.05 ($21 \times 2 = 42$ runs), then a fine pass within ± 0.05 of the coarse winner at steps of 0.01 (~10 more), finding the optimum at 0.01 granularity while staying under ~1 second. Saturation scores are damped on low-saturation images:

$$\begin{aligned}\text{sat_confidence} &= \text{clamp}((\text{sat_std} - 10) / 40, 0.3, 1.0) \\ \text{adjusted_score} &= \text{quality} \times \text{sat_confidence}\end{aligned}$$

An image with a pure black border (found by scanning the 1-pixel perimeter) skips saturation entirely and relaxes border rejection.

Final selection (`_select_best`) prunes, then breaks ties by area:

1. **Quality threshold** — keep candidates whose quality is at least $\max(\text{best} - \text{SELECT_QUALITY_TOLERANCE}, \text{best} \times 0.75)$ — within 0.01 of the maximum (or, on a low-confidence image, within 25%). The tight tolerance stops a clearly-worse quad from winning the area tiebreak.
2. **Peel area floor** (peel only) — when `prefer_larger=False` and more than one candidate remains, drop any below $\text{PEEL_AREA_FLOOR} \times \text{max_good_area}$ (default 0.90). This blocks outliers whose edge cuts through the subject far inside the parent; the practical effect is that one peel finds a tighter inner boundary but not a dramatically smaller one (deeper structure needs more peels).
3. **Area tiebreak** — `prefer_larger=True` (initial) picks the largest region; `prefer_larger=False` (peel) picks the smallest above the floor.

Returns (strategy_name, best_sensitivity, corners). The sweep/selection constants (SENSITIVITY_COARSE_STEP, SENSITIVITY_FINE_STEP, SENSITIVITY_FINE_RANGE, SELECT_QUALITY_TOLERANCE, PEEL_AREA_FLOOR) live at the top of detect.py.

4.9 Coordinate mapping through the peel stack

The peel stack is a list of (image_bgr, corners) tuples — index 0 is the original and its detected corners, index 1 is the rectified image of index 0 and *its* corners, and so on. **Each layer lives in its own coordinate space; there is no mapping between layers.** Peeling in rectifies the current image through its corners, pushes the new image, and re-detects within it with prefer_larger=False. Peeling out simply pops the stack (instant).

Two places do need coordinate mapping, both by composing perspective transforms:

Drawing the overlay. To show the innermost region on the original-image left panel, the deepest corners are mapped *backward* through each level:

```
For k = n-1 ... 0:
    dst_rect = compute_output_size_corrected(...) # aspect-corrected at level 0
    Hk_inv = getPerspectiveTransform(dst_rect, corners_k)
    points = perspectiveTransform(points, Hk_inv)
```

Editing at depth > 0. A point dragged in original-image coordinates is mapped *forward* through all levels to update the deepest corners:

```
For k = 0 ... n-1:
    Hk = getPerspectiveTransform(corners_k, dst_rect)
    point = perspectiveTransform(point, Hk)
```

4.10 Rectified-space nudging

All nudges (corner, edge, whole-rectangle) operate in the *rectified output's* coordinate system, so "move the top edge up" is unambiguous regardless of the quad's orientation in the photo. The mechanism: read the corners in image space → compute the transform to the output rectangle → apply the per-corner deltas in that axis-aligned space → map back with the inverse transform. The three modes:

- **Corner** — only the selected corner's delta is non-zero.
- **Edge** — in rectified space edges are axis-aligned: edges 0/2 (top/bottom) move vertically for perpendicular arrows and shift horizontally for parallel ones; edges 1/3 (right/left) the reverse.
- **Whole-rectangle** — all four corners move symmetrically; up/right expands (+1), down/left shrinks (-1), each corner's delta pointing away from the centroid (TL: -, -; TR: +, -; BR: +, +; BL: -, +). The active region is the inner 50% of the quad; a click outside it but inside the quad selects edge mode instead.

4.11 Detection padding

Before detection, images are padded by replicating edge pixels outward:

```
padded = cv2.copyMakeBorder(image, pad, pad, pad, pad, BORDER_REPLICATE)
```

with pad = 100 (DETECTION_PADDING). Corner coordinates are shifted back by pad afterward. This gives subjects near the image boundary a comfortable margin for the edge detector and the margin score; without it, such detections would be penalized or missed. **Padding is applied only to the original image** (initial detection and

depth 0); on rectified peel results it is skipped, because replicating the edge pixels of a cropped painting would create false boundaries.

4.12 The transform pipeline

Key functions in `transform.py`:

- **`compute_output_size(corners)`** — the natural rectangle size (max of each opposite-side pair; see §4.6).
- **`is_valid_quad(corners)`** — rejects degenerate quads: any two corners within 1 pixel, output width/height under 4 pixels, or contour area under 16 px².
- **`rectify(image, corners)`** — validates, sizes the output, builds destination points `[0,0]`, `[w-1,0]`, `[w-1,h-1]`, `[0,h-1]`, then `getPerspectiveTransform + warpPerspective`.
- **`rectify_full_image(image, corners, aspect_ratio, crop, fill_color)`** — applies the same homography to the *whole* image: compute `H`, transform the four image corners to find the output bounding box, compose a translation so all coordinates are non-negative, cap dimensions at 16384 px, warp with a constant fill, and optionally crop to the largest inscribed rectangle.
- **`_warp_full_image(image, H, crop, fill_color)`** — the shared helper for the full-image and keystone paths (bounding box, translation, 16384 cap, optional inscribed crop).
- **`_crop_inscribed_rect(...)`** — finds the largest axis-aligned rectangle inside the warped (convex) quadrilateral, sweeping vertex-aligned y-values and computing left/right boundaries per scanline.
- **`keystone_homography(line_pairs, w, h)`** — makes line pairs parallel by mapping their vanishing point(s) to infinity. One pair: the minimum-norm solution sending one vanishing point to infinity. Two pairs: the vanishing line through both points is mapped to the line at infinity (affine rectification). A roll correction then makes verticals truly vertical and horizontal truly horizontal. Computed in centered coordinates (image midpoint) for symmetric distortion.
- **`keystone_correct(image, line_pairs, crop, fill_color, scale_x=1.0)`** — computes the keystone homography and applies it via `_warp_full_image()`.

4.13 GUI architecture internals

Class hierarchy. `RectifyMainWindow(QMainWindow)` owns all state, builds the UI, and handles signals. `SourcePanel(ImagePanel)` is the left panel (image + quad overlay); `ImagePanel(QGraphicsView)` is the base zoom/pan display; `CornerHandle(QGraphicsEllipseItem)` is a draggable, clamped corner; `ArrowHandle(QGraphicsPolygonItem)` is a keystone-line endpoint.

Coordinate system. `QGraphicsView` transforms automatically between view (screen) and scene (image) space; corner handles live in scene coordinates, which are image pixels. The `ItemIgnoresTransformations` flag keeps handles a constant on-screen size at any zoom.

Undo/redo. The undo stack stores copies of the 4×2 corners array (negligible memory). An entry is pushed when a drag *begins*, not per pixel of motion. Undo/redo are per-peel-level and cleared on peel in/out.

Cursor contexts. The left panel uses `_find_element_at` to pick a cursor, with corner proximity (`CORNER_SELECT_RADIUS`, in screen pixels converted to scene units so the hit area is zoom-independent) taking priority over edges. Hover: crosshair on a corner, four-arrow cross on an edge, open hand in the center or for panning, plain arrow when the image fits. Active (button or Shift held): closed hand.

Zoom anchoring. Zoom anchors under the cursor when it is over the image (`AnchorUnderMouse`) and centers the image when the cursor is in the gray margin (`AnchorViewCenter`). The scene rect is expanded at load so anchoring works from the first tick; scroll bars are disabled (panning is by dragging).

Settings persistence. `_save_settings()` (from `closeEvent`) writes JSON; `_load_settings()` (from the constructor) restores it. The path is `QStandardPaths.AppDataLocation` with `organizationName` set to the OS username. Image/corner restoration is deferred to `showEvent` (`_restore_saved_image`) so the window is laid out

first; it rebuilds the peel stack by re-rectifying each level from the saved corners. A `_transform_corners_edited` flag prevents floating-point drift when carrying corners back from Transform to Extract.

Before/after. Holding Space sets `_space_held`, and `_draw_result()` shows the original instead of the result; auto-repeat is ignored to avoid flicker.

4.14 Color-theory primer

This primer explains the ideas the color tools rest on (§2.12, §3.4), assuming no prior color science.

The color cast, and the illuminant. A camera records the color of the *light* multiplied by the color of the *surface*. Photograph a white wall under a tungsten bulb and it comes out orange; under shade it comes out blue. That tint is the **color cast** of the **illuminant** (the light). To recover the subject's true color you have to divide the light's color back out.

Neutral references tell you the cast. A *neutral* surface — one that reflects all wavelengths roughly equally, i.e. some shade of gray or white — has no color of its own, so whatever color it shows *is* the light. If you photograph a neutral patch beside your subject and measure it, you have measured the cast directly. That is why you place a gray card or a white sheet in the shot.

White balance (the von Kries method). Once you know the cast, removing it is simple: scale each color channel (red, green, blue) by whatever factor makes the neutral patch read equal across channels. After that division the patch is truly gray, and — assuming the light was the same color everywhere — so is everything else's color relationship. This per-channel scaling is the classic *von Kries* model of color adaptation, and it is exactly what Rectify computes (§3.4).

Reflectance, and "middle gray." *Reflectance* is the fraction of light a surface returns — 0% is perfect black, 100% perfect white. A photographic **gray card** is manufactured to a known reflectance, almost always **18%**, the standard "middle gray." Eighteen percent sounds dark, but human lightness perception is roughly logarithmic, so 18% *looks* about halfway between black and white. A **diffuse white** reference (a white card, clean matte paper) sits near **90%** — not 100%, because real matte surfaces don't reflect everything, and because leaving headroom above it lets genuine highlights (a specular glint, a lamp) stay brighter.

Why the math is done in linear light. The numbers stored in a JPEG are *sRGB-encoded*, a deliberately non-linear curve that gives dark tones more code values (matching perception). Physical light, though, adds linearly, so gains must be computed in **linear light**: Rectify decodes the sampled patch from sRGB to linear, computes the gain, applies it, and re-encodes. A consequence worth remembering: 18% reflectance encodes to about **118** on the 0–255 sRGB scale, *not* 128 — "50% gray" (128) is a common slip that actually corresponds to a much higher reflectance.

Coupled vs. decoupled correction — the real choice. Sampling a neutral fixes the *color*; the open question is what to do with *brightness*:

- **Coupled** (Rectify's **Gray card** and **90% White** modes): map the neutral to a *target reflectance*. Because the target is the same on all three channels, one operation both neutralizes the cast and sets the exposure (the patch lands at the target level). Use this when your reference has a known reflectance — you get correct color *and* a calibrated exposure in one click. Gray card lets you dial the target (your card's rating, default 18%); 90% White fixes it at the diffuse-white standard.
- **Decoupled** (Rectify's **Neutral gray** mode): neutralize the cast but leave the patch at the brightness the camera actually captured, with a *separate* brightness control. Use this when you only want the cast gone and don't want to force an exposure — it preserves highlight headroom (nothing is driven to pure white).

Why three modes, and why 90% White is locked. The genuine axis is coupled vs. decoupled, not "gray vs. white." Gray card and 90% White are the *same* coupled engine at different targets; what keeps 90% White from being merely "Gray card at 90%" is that the Gray card slider caps at 80%, so the two ranges don't overlap and the

diffuse-white anchor has its own one-click mode. Gray reflectances vary (cards come in many ratings, so that target earns a slider); diffuse white is a single standard (so it is a fixed, knob-free target). Making 90% White adjustable would just duplicate Gray card.

Honest limits. This corrects the *illuminant*, faithfully to the file's encoding — but it cannot undo the white balance and tone curve a camera already baked into a JPEG or HEIC (those formats are *display-referred*: already finished pictures, not raw sensor data), and a single global gain assumes the light's color is uniform across the frame. For a painting lit evenly by one source it works very well; under mixed lighting it corrects the cast on average.

Reference: J. von Kries, "Die Gesichtsempfindungen," in *Handbuch der Physiologie des Menschen*, 1905 (the per-channel adaptation model underlying white balance).

4.15 References

- J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.
- U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244–256, 1972.
- D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.
- R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004.
- Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.
- A. Criminisi, I. Reid, and A. Zisserman, "Single view metrology," *International Journal of Computer Vision*, vol. 40, no. 2, pp. 123–148, 2000.
- J. von Kries, "Die Gesichtsempfindungen," in *Handbuch der Physiologie des Menschen*, vol. 3, 1905.