Twenty-Four Into a Hundred and Twenty

Drawing the Frames a Phone's Screen Is Waiting For

Ivor
Independent • *Ivorisnoob@gmail.com*
November 2026

Abstract

Many phones now refresh their screens 120 times a second, and most of the video played on them runs at 24 to 30 frames a second, so a 120 Hz panel shows each film frame five times. Motion interpolation (drawing the frames that were never shot) is the obvious fix. It also has the worst reputation in consumer video: on televisions it shipped switched on, was buried in menus, and visibly broke on hard motion. This essay describes SmoothMotion, an open-source interpolation engine for Android's Media3 player that I wrote for my own apps, and uses it to argue a narrow claim: the reputation belongs to a deployment, not to the technique. Three properties made televisions' interpolation objectionable (it was imposed, it was invisible, and it failed loudly), and none of them is intrinsic. I set out how a 24 fps source becomes a steady 120 fps clock that the display follows by itself; how the engine went through three designs in about a day; why its motion search never runs on a frame wider than 320 pixels; and why most of the engineering is about deciding when not to run. This version adds measurements. Its shipped shaders, run offline on 610 held-out frames from three open films, beat both doing nothing and a plain cross-fade on average on ordinary footage (41.2 dB against 36.1 dB and 39.9 dB at 30 → 60 fps). Its worst frame there is 4.6 dB below a cross-fade, against 29.3 dB below for the first engine I wrote. The per-pixel engine it replaced is about 2 dB more accurate on average but fails far harder at its worst, which is the trade this essay argues for, now with a price on it. Three further tests follow. Against SVP's own method, SmoothMotion is 3.8 dB more accurate on the 60 fps film and 0.8 dB on live action. On fight scenes and scene cuts from *Sintel*, it catches every cut but also treats over a third of the other pairs as cuts, and no design beats a cross-fade by much: SVP leads on PSNR there, by about a decibel, while a perceptual measure, LPIPS, ranks SmoothMotion ahead of it. And feeding the old engine's vectors into the new renderer recovers only 0.2 to 1.1 dB, so the old engine's lead was mostly in how it drew, not in what it found. A twelve-minute device log shows both the first 120 Hz confirmation on a phone and the cadence bug, frame by frame. A section of formulas writes the whole engine down, from the output clock to the renderer, and shows why the search's cost is constant and why the cadence bug could never correct itself. Battery cost is still unmeasured.

Keywords: frame interpolation, motion estimation, mobile GPU, video playback, display refresh rate, Media3, motion smoothing, PSNR, SSIM, LPIPS, SVP

1. Introduction

My phone's screen refreshes 120 times a second. When I watch a film on it, it shows me each frame five times.

That is not a defect. It is what the film is: twenty-four photographs a second, a rate the industry settled on in the late 1920s because it was about the slowest at which the sound on the film's edge still played cleanly, and film stock was expensive [1]. Every frame the phone repeats is a frame that was never shot. The panel could show something new four times out of five, and it does not, because there is nothing new to show.

Drawing that something is a solved problem in one sense and a poisoned one in another. Motion-compensated interpolation has been in televisions for well over a decade, in a separate chip in some phones, and in a desktop program called SmoothVideo Project that a small, devoted audience has run on films for years [2, 3]. It is also the single feature in consumer video that famous directors have recorded public service announcements against [4]. Turning it off has its own industry standard [5].

I wrote an interpolation engine anyway. It is called SmoothMotion; it ships in two Android apps of mine, Koda and OpenStream, and as a standalone library any app built on Google's Media3 player can import [6, 7, 8]. This essay is about what it took to make it, and it makes one argument, stated here so that nobody has to read five sections to find it.

The reputation is about a deployment, not a technique. What made television interpolation objectionable was that it was imposed (on by default, hard to find, renamed by every manufacturer), invisible (nothing on screen said it was running), and loud when it failed (the halos and tearing around fast motion that give the effect away). None of those three is a property of drawing frames between frames. They are choices, and they can be made the other way: off by default, visible about what it is doing, and quiet when it is wrong. Most of this essay is about the third, because it is the hard one.

I am not going to argue that films should be smoothed. Some people want it and most filmmakers do not, and I think the filmmakers are right about their own films. The claim is narrower: it is the viewer's call, on the viewer's device, and it can be offered honestly.

A note on method, as in my first essay [9]. Numbers from

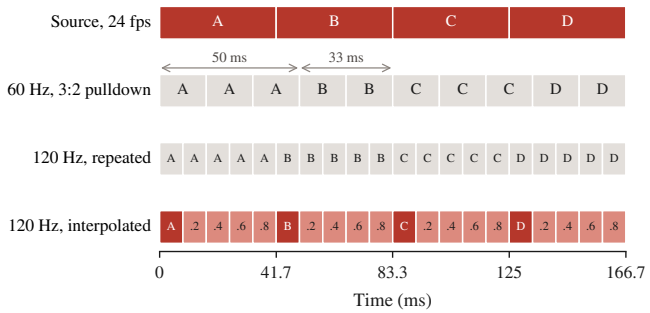


Figure 1. Four film frames (A to D) on three displays. At 60 Hz, 3:2 pulldown alternates 50 ms and 33 ms holds. At 120 Hz every frame is held five refreshes. Interpolated, each real frame is still shown untouched once, and the four ticks between are drawn at phases 0.2, 0.4, 0.6 and 0.8 of the way from one to the next.

my own code and commits are cited to the commit, so they can be checked. Numbers from devices are labelled with the device. The first version of this essay listed four things as unmeasured, and promised that the version carrying them would say so on its first page. This is that version, and it carries two of the four, plus the two gaps the second version found in itself. Section 7 measures the engine's quality against held-out frames and compares it with every earlier design, with SVP's own method, and with two baselines, on 610 frames from three open films, one of them chosen for its scene cuts and fights, using the exact shaders that ship. It scores them with a perceptual measure as well as the usual two, and tests whether better vectors would close the gap to the engine I replaced. Section 6 draws the device bug report from its log rather than from memory. Battery and thermal cost remain unmeasured, and Section 9 says so.

2. THE GAP

2.1. What a screen does with a film

A display can only show a frame when it refreshes. Video at 24 fps on a 60 Hz screen does not divide evenly: 60 refreshes over 24 frames is 2.5 each. The standard answer is 3:2 pulldown, which holds one frame for three refreshes and the next for two, so frames alternate between 50 ms and 33 ms on screen. That unevenness is judder, the stutter you see on a slow pan, and Android's own documentation names pulldown as its cause [10].

A 120 Hz screen fixes the arithmetic. 120 divides evenly by 24, by 30 and by 60, so every frame of a film is held exactly five refreshes and every frame of a 30 fps video exactly four. Figure 1 shows the three cases. The holds are even, which removes judder, but they are still holds: the image sits still for 41.7 ms, then jumps. On a phone held at arm's length, panning shots step.

Interpolation fills those four ticks. The bottom row of Figure 1 is the target: the decoded frame unaltered on every fifth refresh, and four new frames between, each placed at its own fraction of the way to the next.

2.2. Why phones, and why now

Two things changed. High refresh rates moved from gaming monitors into ordinary phones, so the spare refreshes now exist in most pockets. And phone GPUs became general-purpose enough to do the work: the engine described here needs OpenGL ES 3.2, with compute shaders and buffer atomics, which current Adreno and Mali GPUs provide [7].

Phone makers noticed before I did. OnePlus announced a dedicated motion-compensation chip in 2020 to raise 30 fps video to 120 on its phones, and Pixelworks' X7 visual processor, which does the same kind of job in hardware, went into the China version of the OnePlus 11 and into Xiaomi's Redmi K60 Ultra in 2023 [11, 12, 13]. Those are real products. They are also closed, tied to specific phones, and switched on in the manufacturer's settings rather than the app's. On a phone without one of those chips, a video app had no way to do this at all.

3. THE REPUTATION

Before arguing that interpolation can be done well, it is worth being exact about why it was done badly, because the bad version is what nearly everyone has seen.

3.1. It was built to fix a different problem

Televisions did not adopt interpolation to make films smoother. They adopted it to fix their own panels. An LCD holds each image for the whole refresh, and an eye tracking a moving object across a held image smears it; this is sample-and-hold blur [14]. Doubling the refresh rate to 120 Hz halves the hold, but a 60 Hz broadcast has only 60 images to show, so the set had to make up the rest. Interpolation was one of the two fixes on offer; the other was flashing black between frames [14].

So the feature arrived as a panel correction, enabled by default because the panel needed it, and applied to everything the set displayed, films included, by people whose goal was sharper sports rather than better cinema.

3.2. What people actually objected to

Two different complaints get filed under "the soap opera effect", and they deserve to be separated.

The first is aesthetic. Twenty-four frames a second is part of how film looks, and adding motion makes it look like video. This is not a technical judgement. When Peter Jackson shot *The Hobbit* at a real 48 fps in 2012, with no interpolation at all, critics reached for the same words: the shimmer of a soap opera, a high-end home movie, sets and costumes whose phoniness became obvious [15]. The objection survives perfect frames. It is about what the viewer expects film to be, and it is a legitimate preference.

The second complaint is about artefacts. Interpolation that guesses motion wrongly produces visible errors: halos around moving edges, background tearing behind a fast object, text that warps as it scrolls. These are failures, and on the hard cases (fast motion, repeating patterns, scene cuts) cheap en-

gines fail a lot. Figure 7 shows one of mine doing it.

The campaign that followed attacked deployment rather than either complaint directly. In December 2018 Tom Cruise and Christopher McQuarrie recorded a short video from the set of *Top Gun: Maverick* asking viewers to turn motion smoothing off, objecting that it shipped enabled and that the setting was buried under names that varied by brand [4]. In August 2019 the UHD Alliance announced Filmmaker Mode, a single setting that turns off interpolation and the rest of a television's processing, built with input from hundreds of filmmakers [5].

Look at what those two responses targeted. Not the algorithm: the default, and the discoverability. That is the opening this essay walks through.

3.3. The honest position

If the aesthetic objection is right for a given viewer, no engineering answers it, and the correct behaviour is to stay off. That is why SmoothMotion is off by default in both apps, and Koda turns it on only through a warning [6, 8]. The artefact objection is different. It is an engineering claim, it can be measured, and Sections 6 and 7 are largely the story of failing it, fixing it, and then measuring how far the fix goes.

4. The System

SmoothMotion is about 4,000 lines of Kotlin and GLSL across three modules, plus about 500 lines of tests: a core with the GPU engine and the timing rules, which has no dependency on Media3; a Media3 module that plugs the engine into the player's video effects pipeline; and a small UI module [7]. Figure 2 shows the path a frame takes.

4.1. A clock, not midpoints

The first version did the obvious thing: between every two decoded frames, draw one frame halfway, doubling 30 fps to 60 [16]. It was replaced within a day, and the reason is the most useful design fact in the project.

Media3's video sink does not take the display rate from the app. It runs its own estimator over the timestamps of the frames it is handed, and once those settle into a steady rate it asks the display for that rate through Android's `Surface.setFrameRate` [7, 10]. An output that is regular at 120 fps therefore raises a 120 Hz phone from its idle 60 Hz by itself. An output that is irregular (say, midpoints plus every real frame, for a source whose rate does not divide the screen's) never settles, and the display stays where it was.

So the output is a clock. It ticks at the target rate in media time, computed from an origin so that a fractional step never drifts. When the source interval divides into whole ticks within 1.5% (24 into 120, 30 into 60), the step snaps to that division, so every fifth or second tick lands exactly on a decoded frame, which is copied out untouched [7]. A tick between two decoded frames, A and B, is handed to the engine with its phase t, the fraction of the way from A to B.

The clock also refuses work it cannot afford. Frames larger than about 2.2 megapixels per output frame at 120 fps (a little beyond 1080p) step the target down to 90, then 60; a slow playback speed that would demand more than eight outputs per decoded frame halves the rate instead; and a source already within 15% of the screen's rate passes straight through, because a 60 fps video on a 60 Hz screen has nothing to gain [7].

4.2. Finding the motion

To draw a frame at phase t the engine needs to know where each part of the picture moved between A and B. It finds out by block matching on the GPU: each frame is reduced to normalised luminance, built into a pyramid of shrinking copies, and divided into 8×8 blocks. At the coarsest level each block searches ±8 pixels for its best match in the other frame; the level under it re-searches ±2, and every finer level refines ±1 around the best of up to seven predictions (no motion, the coarser level's answer and its four neighbours, and the previous frame pair's answer) [7]. The search runs in both directions, A to B and B to A, and a 3×3 median over neighbouring vectors removes outliers. The median picks one of the neighbours' actual vectors rather than taking the median of each component separately, which would invent vectors no block had and break thin objects apart [17].

The decision that made this affordable is that the finest level of the pyramid is never wider than 320 pixels on its long side, whatever the video's resolution [18]. Motion in a film is coherent at that scale; the engine does not need a vector per pixel to know that the background is panning left, because the frame it draws interpolates the vectors smoothly anyway (Section 4.3). Searching the full-resolution pyramid had cost an Adreno 613, a budget GPU, about 310 ms per 480p frame pair and about 1.4 s at 1080p, against a budget of 33 ms for a 30 fps source. Capped at 320 pixels, the search took about 23 ms at any size [18]. Figure 3 puts these on one axis. Section 7.4 reproduces the shape of this result on another GPU and measures what the cap costs in quality.

4.3. Drawing the frame

Having vectors is not the hard part. Turning vectors into a frame that does not break is.

The renderer is a port of open-svpflow, an Apache-licensed reimplementation of the method behind SmoothVideo Project, rewritten from Rust into OpenGL ES compute and fragment shaders [19, 20]. Its central property is that it makes no decisions per pixel. Every output pixel is one formula over two samples: frame A read along the backward vector field at phase t, and frame B read along the forward field at $1-t$, with both fields sampled bilinearly between block centres, then blended by time [17]. Three corrections are layered on that formula, all of them smooth:

- a *coverage* mask, built by splatting each block to the output moment with atomic adds, which swaps in the other frame's sample where content has just been uncovered and one sample has nothing true to show;

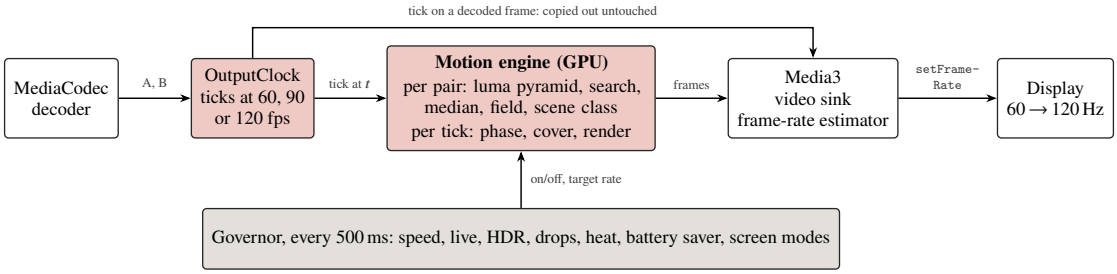


Figure 2. The path of a frame. The effect sits inside Media3's own pipeline, between the decoder and the video sink. A tick between two decoded frames, A and B, is drawn by the engine at phase t; a tick that lands on a decoded frame copies it out untouched; any engine failure switches that player to plain pass-through. Because the sink estimates the frame rate from the timestamps it receives and asks the display for it, a steady 120 fps output is enough to raise the panel from 60 to 120 Hz without the app requesting anything [7].

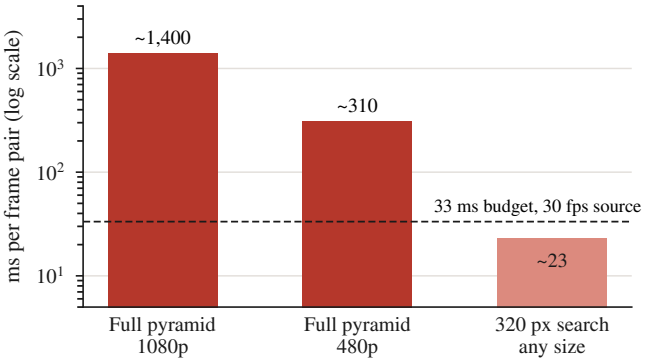


Figure 3. Motion search time per frame pair on an Adreno 613, before and after the search was moved to a copy of the frame at most 320 px wide. Figures are as reported in the commit that made the change [18]; they are developer measurements on one device, not a benchmark. Figure 9a repeats the experiment on a desktop GPU.

- a *bad-area* mask that fades poorly matched blocks toward a plain cross-fade, so a block the search could not explain is blurred rather than misplaced;
- on pairs classed as hard, a clamp that keeps each pixel between its two samples, so it can never take a colour that neither frame contains.

Each pair is also sorted into classes (fine, hard, or a scene cut) from how many blocks match poorly, weighted by brightness. A cut is never interpolated: the tick shows whichever real frame is nearer in time. Hard pairs are re-timed toward real frames, so the engine spends less of the screen's time on frames it is unsure of [17, 19].

The consequence is the property the whole design was chosen for. Where motion is too hard to follow, the picture goes soft. It does not tear. Section 7 measures how true that is, and what it costs.

4.4. What it costs to run

Per frame pair, the engine builds two luminance pyramids, searches both directions, filters, restates the vectors and classifies the pair; per output tick, it computes the phase, splats coverage and renders. Stages that do not depend on each other are queued together, each stage ends in a single memory barrier carrying only what the next stage reads, and the one value read back to the CPU, the scene class for the log, comes back behind a fence rather than by waiting for the GPU [17]. A failure anywhere in the engine, on any GPU, turns interpolation off for that player and plays the video normally [7].

5. THE MATHEMATICS

Everything above can be written down in a page of formulas, and writing it down is the quickest way to see why the failures in Section 6 happened. The notation is the engine's: a source frame interval T, an output step Δ, a phase $t \in [0, 1]$ between decoded frames A and B, 8×8 blocks ($B = 8$) on the search's finest level, and images $A(\mathbf{x})$, $B(\mathbf{x})$ in RGB. Every constant below is the one in the shipped code [17, 7].

5.1. The clock

Output ticks are computed from an origin τ_0 rather than accumulated, so rounding never drifts:

$$\tau_k = \tau_0 + k\,\Delta, \qquad \Delta = \frac{s}{f_{\text{out}}}, \tag{1}$$

where s is the playback speed and f_{out} the target rate (60, 90 or 120). The step snaps to a whole division of the source interval when one is close:

$$n = \operatorname{round}\!\left(\frac{T}{\Delta}\right), \qquad \Delta' = \frac{T}{n} \quad \text{if} \quad \frac{|n\Delta - T|}{T} \le 0.015. \tag{2}$$

A 23.976 fps film has $T = 41.708$ ms; at 120 fps, $T/\Delta = 5.005$, so $n = 5$, $\Delta' = 8.342$ ms, and every fifth tick lands exactly on a decoded frame. Each tick between A and B is drawn at its phase,

$$t_k = \frac{\tau_k - \tau_A}{\tau_B - \tau_A} \in \{0,\ 0.2,\ 0.4,\ 0.6,\ 0.8\} \ \text{ for } n = 5, \tag{3}$$

and the tick with $t_k = 0$ copies A out untouched.

5.2. The search

The search compares not raw brightness but luminance normalised against its neighbourhood, so that a fade or an exposure change still matches. With $Y = 0.299R + 0.587G + 0.114B$, and μ and σ the mean and standard deviation of nine taps spaced 1.5 texels apart,

$$N(\mathbf{x}) = \operatorname{clamp}\!\left(0.5 + 0.18\,\frac{Y(\mathbf{x}) - \mu(\mathbf{x})}{\sigma(\mathbf{x}) + 0.02},\ 0,\ 1\right). \tag{4}$$

Table 1. Pixel comparisons per frame pair from Equation (8), both search directions, and the measured search time on an Intel UHD 630 (Section 7.4).

	320 px cap		Full frame	
Video	comparisons	ms	comparisons	ms
480p	7.2 M	5.3	40.4 M	13.7
720p	7.2 M	3.9	90.1 M	31.2
1080p	7.2 M	3.6	203.3 M	70.2
2160p	7.2 M	4.3	809.7 M	285.9

Each block is matched over a 12×12 window W (the block plus a 2-pixel margin) by mean absolute difference, with a penalty for straying from the predicted vector $\mathbf{p}$ (the parent level's answer, doubled, or the previous pair's):

$$C(\mathbf{v}) = \frac{1}{144}\sum_{\mathbf{x}\in W}\left|N_A(\mathbf{x}) - N_B(\mathbf{x}+\mathbf{v})\right| + \lambda\,\|\mathbf{v}-\mathbf{p}\|, \qquad (5)$$

with $\lambda = 0.006$ (SVP's `penalty.lambda`). The best of seven predictions $\mathbf{v}_0$ is chosen first; then every offset with $\|\mathbf{d}\|_\infty \le r_\ell$ around it is measured, with $r_\ell = 8$ at the coarsest level, 2 at the next and 1 elsewhere. The integer winner is refined to sub-pixel precision by fitting a parabola through its cost S_0 and its neighbours' $S_\pm$ on each axis:

$$\delta = \operatorname{clamp}\!\left(\frac{1}{2}\,\frac{S_- - S_+}{S_- - 2S_0 + S_+},\ -\tfrac{1}{2},\ \tfrac{1}{2}\right). \qquad (6)$$

The 3×3 vector median then replaces each vector with the neighbour closest to all the others,

$$\mathbf{v}^* = \arg\min_{\mathbf{v}_i\in\mathcal{N}} \sum_{\mathbf{v}_j\in\mathcal{N}} \|\mathbf{v}_i - \mathbf{v}_j\|^2, \qquad (7)$$

which can only ever return a vector some block actually had.

5.3. Why the cap makes the cost constant

Counting pixel comparisons makes the 320-pixel decision exact. A level ℓ of $w_\ell \times h_\ell$ pixels has $\lceil w_\ell/8\rceil\lceil h_\ell/8\rceil$ blocks; each is compared at 7 predictions and $(2r_\ell+1)^2$ offsets, each comparison covering 144 pixels, in both directions:

$$K = \sum_\ell 2\left\lceil\frac{w_\ell}{8}\right\rceil\left\lceil\frac{h_\ell}{8}\right\rceil\left(7 + (2r_\ell+1)^2\right)\cdot 144. \qquad (8)$$

With the finest level fixed at 320×180, K is 7.2 million comparisons per frame pair for any 16:9 video. Searching from the full frame instead, K grows with the pixel count: 40 million at 480p, 90 million at 720p, 203 million at 1080p and 810 million at 2160p (Table 1). Section 7.4 timed both on a desktop GPU; the measured times track Equation (8) closely, at about 0.35 ms per million comparisons (2160p cost 4.08 times 1080p's time, against 3.98 times its comparisons).

5.4. The frame

The renderer samples the two vector fields bilinearly between block centres at the pixel's own position, so a field varies smoothly across the picture. With $\mathbf{v}_b$ the backward field (B to A) and $\mathbf{v}_f$ the forward field (A to B), the two motion-compensated samples are

$$a = A\big(\mathbf{x} + t\,\mathbf{v}_b(\mathbf{x})\big), \qquad b = B\big(\mathbf{x} + (1-t)\,\mathbf{v}_f(\mathbf{x})\big). \qquad (9)$$

On an easy pair (SVP's algorithm 21), each sample is swapped toward the other by its coverage mask, c_A or $c_B \in [0,1]$, then the two are blended by time:

$$I_{21} = (1-t)\big[(1-c_A)\,a + c_A\,b\big] + t\big[(1-c_B)\,b + c_B\,a\big]. \qquad (10)$$

On a hard pair (algorithm 13), the result is the per-channel median of the two samples and the plain cross-fade, so a pixel can never take a colour that neither frame offers:

$$I_{13} = \operatorname{med}\big(a,\ b,\ (1-t)A(\mathbf{x}) + t\,B(\mathbf{x})\big). \qquad (11)$$

Finally the bad-area mask m fades the result toward a safe fallback F, the cross-fade for algorithm 13 and the nearer real frame for algorithm 21:

$$I = (1-m)\,I_{\text{alg}} + m\,F, \qquad m = \min\!\left(1,\ \frac{S}{2400}\right), \qquad (12)$$

where S is the block's sum of absolute differences in 8-bit luma at its vector, which reaches 2,400 when the 64 pixels of a block are off by an average of 37 levels, the point at which the engine stops trusting the block at all. All of Equations (9) to (12) are continuous in $\mathbf{x}$, because every quantity in them is interpolated between block centres. That continuity is what "fails soft" means in mathematical terms: a wrong vector moves a neighbourhood smoothly, and cannot make two adjacent pixels disagree.

5.5. Scene classes

Each pair is classified from its forward blocks' scores, weighted down in bright scenes where differences are larger for the same visual change:

$$\begin{aligned} \sigma_i &= \left\lfloor \frac{255\,S_i}{L_i} \right\rfloor, \\ L_i &= \max\!\left(20,\ \left\lfloor 255\Big(\frac{\bar{Y}_{A,i} + \bar{Y}_{B,i}}{255}\Big)^{1.5}\right\rfloor\right), \end{aligned} \qquad (13)$$

where $\bar{Y}_{A,i}$ and $\bar{Y}_{B,i}$ are the block's mean 8-bit luma in each frame. Ignoring a 4% border, blocks with σ_i below $200\cdot B^2/32$ are near-still, and up to two thirds of the grid of those are not counted. Of the N counted blocks, the pair is a cut if at least $\lfloor 0.2N\rfloor$ exceed $4000\cdot B^2/32$; hard (class 2 or 1) if that many exceed $2800\cdot B^2/32$ or $1600\cdot B^2/32$; and fine otherwise.

5.6. Drops, and the cadence bug in numbers

With D frames dropped, E emitted and F decoded, the governor's drop rate and the naive one differ by exactly the interpolation ratio $k = E/F = f_{\text{out}}/f_{\text{src}}$:

$$\rho = \frac{D}{E}, \qquad \rho_{\text{naive}} = \frac{D}{F} = k\,\rho. \qquad (14)$$

At 30 into 120, $k = 4$, so a true 5% reads as 20% (Figure 4). The log's measure of cadence follows from the same counters: with E emitted and G drawn, untouched decoded frames per second are

$$R = \frac{d(E-G)}{d\tau}, \qquad R = f_{\text{src}} \text{ when locked.} \tag{15}$$

The bug itself is two tolerances that do not overlap. A running mean of n intervals that takes in one dropped frame's gap of $2T$ reads $T(1+1/n)$. The clock restarts only when its estimate moves by more than 3%, but snaps only when it is within 1.5%. Any error ε with

$$0.015 < \varepsilon < 0.03 \tag{16}$$

leaves a clock that neither snaps nor restarts, and stays wrong for the rest of the video. The log's 122 fps on a 120 Hz screen is $\varepsilon = 122/120 - 1 = 1.7\%$, inside that band; the screen shows 120 of the 122 frames a second and discards the other two. The fix narrows the gap from both sides: outlying intervals are left out of the mean unless three arrive in a row, the clock re-anchors on every pair, and $f_{\text{out}} \le 1.001\, f_{\text{screen}}$ [29].

5.7. The measures used in Section 7

For a rebuilt frame $\hat{I}$ and the real frame I it replaces, both $H \times W$ with 3 channels,

$$\begin{aligned} \text{PSNR} &= 10 \log_{10} \frac{255^2}{\text{MSE}}, \\ \text{MSE} &= \frac{1}{3HW} \sum_{\mathbf{x},c} \big(\hat{I}_c(\mathbf{x}) - I_c(\mathbf{x})\big)^2. \end{aligned} \tag{17}$$

Each 6 dB of PSNR halves the root-mean-square error; the 2 dB between Engine 2b and SmoothMotion means SmoothMotion's RMS error is about 26% larger on average ($10^{2/20} = 1.26$), and the 12 dB between their worst frames is a factor of four ($10^{12/20} = 3.98$) the other way. SSIM compares local means μ, variances σ^2 and covariance σ_{xy} of luma over 7×7 windows [36]:

$$\text{SSIM} = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{18}$$

with $C_1 = (0.01 \cdot 255)^2$ and $C_2 = (0.03 \cdot 255)^2$, averaged over the frame. Neither measure knows what a person notices. LPIPS [35] is a learned one: both frames are passed through a fixed image network (AlexNet), and the distance is a weighted sum, over its layers l, of how far apart their unit-normalised activations $\hat{y}^l$ are,

$$\text{LPIPS} = \sum_l \frac{1}{H_l W_l} \sum_{h,w} \big\| \mathbf{w}_l \odot \big(\hat{y}^l_{hw}(\hat{I}) - \hat{y}^l_{hw}(I)\big) \big\|_2^2, \tag{19}$$

with weights $\mathbf{w}_l$ fitted to human judgements of which of two distortions looks closer to a reference. Lower is better; 0 is identical. It is scored on the whole frame, halved by 2×2 averaging when the frame is wider than 960 px. Where a frame is rebuilt exactly (only in Sintel's fade to black), PSNR is infinite, so it is capped at 60 dB per frame. The baselines are $\hat{I} = A$ (repeat) and $\hat{I} = (1-t)A + tB$ (cross-fade), and every "gain" in Section 7 is $\text{PSNR}_{\text{method}} - \text{PSNR}_{\text{cross-fade}}$ for the same pair.

Table 2. How the engine reached its present design, from Koda's history [6]. Times are IST. "Engine" numbers are the ones used in Section 7.

When	Change, and what forced it
27 Sep, 21:59	Engine 1: midpoint frames, 24–30 into 48–60 fps, off by default
28 Sep, 00:03	Drop check rebuilt: batched drop reports were switching it off
28 Sep, 12:11	Output clock up to 120 fps; Engine 2b, a compute engine after FSR 3's optical flow, with a reworked fragment Engine 2a as fallback
28 Sep, 18:40	Renderer replaced by a port of SVP's method: per-pixel vectors tore into blocks
28 Sep, 19:50	Every tick drawn: at 30 into 60 nothing had been interpolated
28 Sep, 23:00	Search capped at 320 px: a budget GPU took 310 ms per pair
30 Sep, 10:35	Cadence fix after a Mali-G615 report
1 Oct	Ships in Koda 5.1 [31]; published as a library

6. What Broke

The first essay I published argued from other people's measurements [9]. This one has the advantage that I can show my own mistakes with timestamps. The engine reached its present design between 21:59 on Sunday 27 September 2026 and 23:00 the next day, over the commits in Table 2, and the four failures below shaped it more than any plan did.

6.1. Deciding per pixel

The second engine, written after studying AMD's FSR 3 frame generation, chose a motion vector for each pixel by how well colours agreed along it [21, 22]. On easy footage it looked good. Where motion was hard (repeating textures, motion blur, fast movement), neighbouring pixels chose differently on the review clips I was watching, and the picture broke into blocks: exactly the artefact the reputation is made of [23].

The fix was not a better per-pixel choice. It was to stop choosing. SVP's formula reads every pixel from the same two smooth fields, so neighbouring pixels can only differ smoothly, and a region the engine cannot explain degrades into a blur instead of a mosaic. Fed SVP's own vectors, the port matched open-svpflow's CPU renderer at 43.2 dB on easy pairs and 44.7 dB on hard ones (in luma), against a 46.7 dB ceiling set by the RGB-to-YUV round trip on real frames [23, 17].

That decision was made by eye, on a handful of clips. Section 7 tests it against held-out frames, and the answer is more interesting than the commit message: on average, the engine I threw away was the more accurate one.

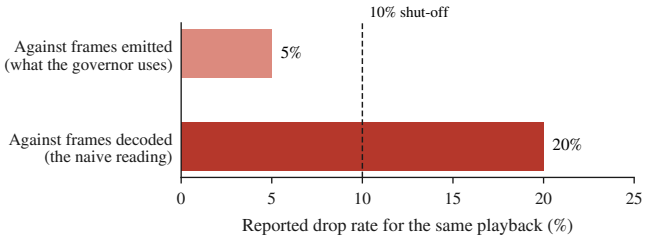


Figure 4. One playback, a 30 fps source at 120 fps output with 5% of output frames dropped, measured two ways. Divided by decoded frames, the rate is inflated by the 4× interpolation ratio and crosses the 10% shut-off. Source: [27].

6.2. Drawing nothing

After the rebuild, the engine was running its whole motion search on every frame pair and, at 30 into 60 fps, drawing nothing with the result [24]. The original rule took the tick nearest the middle of a pair and showed the next real frame for every other tick; at 30 into 60 the only tick between two frames was already a real frame by that rule, and at 30 into 120 it drew one frame and repeated the next real frame three times.

Nothing crashed. The feature reported itself active, the GPU was busy, and the video stepped exactly as before. It is the most instructive bug in the project, because the evidence that the engine was working (status, load, a higher output rate) was all present and all beside the point. The fix draws every tick of an easy pair at its own phase, which is the bottom row of Figure 1. The same commit narrowed the search radius at every level but the coarsest; until then 720p had been the most expensive size, at about 1.6 billion comparisons per frame pair [24].

6.3. Counting drops

The feature has to stop when a phone cannot keep up, and the first version decided that from the dropped-frame reports Media3 sends. Those arrive in batches, when the video renderer stops, and changing video quality stops it. So the drops of a 1080p 60 fps stream, which was never interpolated, arrived the moment a 30 fps quality was picked, and switched the feature off for the very stream it was about to work on [25].

The rebuilt check samples counters every half second instead, counts only periods in which frames were drawn, ignores the first six seconds after a start, seek or quality change, and acts only on more than 10% dropped for about ten seconds [7, 26]. It also had to fix its denominator. With interpolation running, Media3 counts late output frames as drops while counting only decoded frames as rendered, so dividing one by the other inflates the rate by the interpolation ratio: a 30 fps source at 120 fps output dropping 5% of its frames reads as 20% [27]. Figure 4 shows the difference, which is the difference between a feature that works and one that turns itself off.

6.4. Losing the cadence

The last failure came from a user's bug report rather than my own testing, and it is the one I have not finished with. The report came from a Xiaomi phone with a Mali-G615 MC6 GPU, playing a 1080p 23.98 fps video at "Up to 120" on Koda 5.0, and it carried the engine's own log: one line every ten seconds with cumulative counts of frames emitted, frames drawn, frame pairs and drops [28]. Subtracting drawn from emitted gives the number of ticks that showed a decoded frame untouched. Figure 5 plots every interval in the log.

Two things are true of that log at once. The first is the result the first version of this essay quoted: over the first 10.7 minutes of logged intervals, the phone emitted 78,361 frames and dropped 345, or 0.44%, and the display followed the output from 60 to 120 Hz with nothing asking it to. That was the first device confirmation that Media3's estimator raises the panel's rate by itself. The second is the bug. At 6.7 minutes a burst of 314 drops in a minute arrived. Afterwards the output still ran at 120, the drop count went back to almost nothing, and by every number the governor watches the feature was healthy; but untouched decoded frames fell from 24.0 to about 5.8 a second. The original frames themselves had been replaced by interpolations. Thirty seconds later the clock restarted at 122 fps on a 120 Hz screen, which discards about two frames a second without counting them as drops [29].

The cause was arithmetic. The source rate was a plain running average of frame intervals, so the gap a dropped frame leaves went into it, and that gap is exactly when the clock restarts. It restarted on a step the video did not have. Because restarting required a 3% change and snapping only tolerated 1.5%, nothing ever brought it back. The fix leaves outlying intervals out of the rate unless three arrive in a row, re-anchors the clock on every frame pair so drift cannot accumulate, and never lets a snapped step run more than 0.1% faster than the screen [29]. It also draws fewer frames rather than more: about 96 a second at 24 into 120, against the 116 of the broken state.

The lesson here is the same as in "Drawing nothing", in a sharper form. A drop rate of 0.44% is a true number from a session in which the feature was quietly wrong for half the time. The governor's counters could not see it, because the defect was not in how many frames arrived but in which ones. That fix is in the code and in the library. As of this writing it has not been confirmed on a device, and Koda's roadmap still lists the bug as open for that reason [30].

7. Measuring It

The first version of this essay listed "quality against held-out frames" as pending, described the standard method, and called it the next thing to run. This section runs it.

7.1. Method

The test is the textbook one. Take a clip, delete frames, have the engine rebuild them from their neighbours, and compare each rebuilt frame with the real one it replaces. I used two tests. In the "2×" test every other frame is deleted and rebuilt

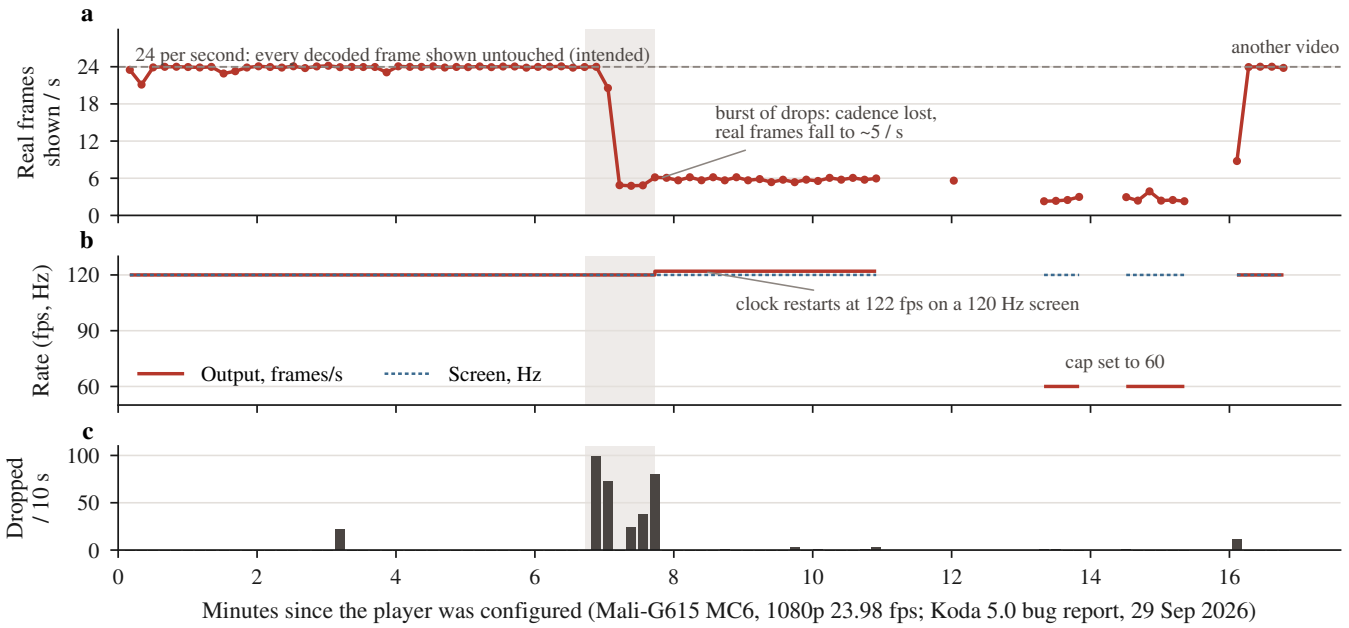


Figure 5. The Mali-G615 bug report, every ten-second interval of the engine's own log [28]. (a) Decoded frames shown untouched per second: emitted minus drawn. (b) Output rate and the screen's reported rate. (c) Frames dropped per interval; the grey band is the burst. For six and a half minutes the cadence is exact: 24.0 real frames and 95.9 drawn frames a second, 120 out, with the display raised from 60 to 120 Hz by the output alone. After the burst only about 5.8 real frames a second survive, and the restarted clock runs at 122 fps on a 120 Hz screen. The stretch capped at 60 is 24 into 60, which does not divide, so few ticks land exactly on a real frame there by design. A different video, in a fresh player, returns to 24.

at $t = 0.5$; on a 60 fps clip that is exactly a 30 fps source drawn into 60. In the "4×" test three frames in four are deleted and rebuilt at $t = 0.25$, 0.5 and 0.75, which gives four ticks per pair, as at 30 into 120, but with twice the motion between the frames the engine sees.

The footage is from three open films, chosen because their frames can be reproduced in a paper. *Big Buck Bunny* has an official 1080p release at a true 60 fps [32]; I took fourteen segments of 17 frames spread across the film, from a near-still meadow to fast camera pans and the scrolling end credits, which together give 112 rebuilt frames in the 2× test and 168 in the 4×. *Tears of Steel* adds five 24 fps live-action clips at 854×356 [33] (20 and 30 rebuilt frames). Neither film's segments contain a scene cut, so the third set was chosen to be hard. *Sintel* is a 24 fps animated film at 1920×818 with sword fights, chases, fire and a dragon [34]. I scanned all 21,312 of its frames for cuts and motion and took fourteen 17-frame segments from the story: seven around a cut and seven with the fastest motion that had none. Checking them by eye at full size found cuts the scan had missed, so the final labels are hand-checked: ten of the fourteen segments hold a cut (one holds two, so eleven cuts in all), one is a flash to white that fades to black, and three are uncut fast motion. That gives another 112 and 168 rebuilt frames, eleven of each set spanning a cut. Two baselines bracket the engines: repeating frame A, which is what a 120 Hz phone shows without interpolation, and a plain cross-fade of A and B weighted by t, which is the cheapest thing that could be called interpolation.

The engines are the four designs from Table 2, each run from its own commit: Engine 1, the midpoint engine that first shipped [16]; Engines 2a and 2b, the fragment and per-pixel compute engines of the next morning [22]; and the current SmoothMotion engine, from the published library [7]. No shader was rewritten for the test. A Python harness reads each engine's GLSL out of its Kotlin source, changes only the version line and precision qualifiers for desktop OpenGL 4.3, and makes the same calls the engine makes, with the same texture formats, dispatch sizes, stage order and barriers, on an Intel UHD Graphics 630. For the older engines I reused the harnesses written at the time, after checking that they reproduced their recorded scores on a Tears of Steel clip (to 0.01 dB for Engines 1 and 2a, and 0.13 dB for 2b, whose working copy on the day differed slightly from the commit). SVP itself is run through open-svpflow, the open reimplementation SmoothMotion's renderer was ported from [19], as prebuilt VapourSynth plugins, on the same source frames and at the same rates. Quality is reported as PSNR over RGB, SSIM over luma with a 7×7 window [36], and the perceptual LPIPS (Section 5.7). Each test's mean is taken over clips, so that a long, easy segment cannot dominate.

Two caveats belong here rather than at the end. This is a desktop GPU running mobile shaders; precision and filtering may differ slightly on a phone. And PSNR and SSIM are measures of distance from the real frame, not of what a viewer sees; they are the right tool for comparing engines and the wrong one for deciding whether to watch a film smoothed.

7.2. Better than doing nothing

The first question is whether the engine helps at all, and it does. Table 3 and Figure 6a give the summary. On Big Buck Bunny at $30 \rightarrow 60$, SmoothMotion's rebuilt frames average 41.2 dB, against 36.1 dB for repeating the frame and 39.9 dB for a cross-fade. The gap is larger on the structural measure: SSIM of 0.985 against 0.932 and 0.938, because a cross-fade gets the average brightness of a moving edge right while drawing it twice. The ranking holds in all four conditions. On every test SmoothMotion beats repeating the frame on all but a few percent of frames (6.2% at 2× on Big Buck Bunny, mostly near-still shots where a repeat is almost perfect).

The two oldest engines are a different story, and an uncomfortable one. Engine 1, the first version committed behind its warning, averages 31.6 dB on Big Buck Bunny at 2×. That is not only below a cross-fade but 4.5 dB below *repeating the frame*; 77% of its frames were further from the truth than doing nothing would have been, and its worst frame scored 11.7 dB. Engine 2a, the reworked fragment engine kept as a fallback, is little better on that footage. Figure 7 shows what those numbers look like. On the end credits fading in, Engine 1 tears the lettering apart; on live-action Tears of Steel its failures are milder but still put it below a cross-fade on 95% of frames. That was the engine in the codebase for about fourteen hours. The reputation is not a myth, and the first draft of my own engine earned it.

7.3. The engine I replaced was more accurate

The comparison the first version of this essay could not make is between the engine I replaced on 28 September and the one that replaced it. The commit message says the per-pixel engine "tore into blocks" where motion was hard [23]. On the 330 frames from the first two films, it is the most accurate design I have built. Engine 2b averages 43.2 dB on Big Buck Bunny at 2× to SmoothMotion's 41.2, and leads in every condition of Table 3, by between 0.4 and 2.0 dB. Figure 8 shows where the gap comes from. On near-still footage the two engines are within a decibel of each other and of a cross-fade. On the camera pans and the credit scroll, where rebuilding the frame matters most, both gain enormously over a cross-fade (SmoothMotion by 7 to 11 dB), but Engine 2b gains 15 to 19 dB. Per-pixel vectors follow rigid motion and fine text more exactly than vectors interpolated between block centres can, and Figure 7 shows the difference on the scrolling credits.

What Engine 2b does not have is a floor. Figure 6b plots every rebuilt frame of the 2× test against the cross-fade of the same pair. SmoothMotion's frames sit in a narrow band: its median frame is within a fraction of a decibel of a cross-fade, its gains reach 11 dB, and its worst frame is 4.6 dB below one. Engine 2b's band is wider in both directions; its worst frame is 16.6 dB below a cross-fade, on the same fading credits where Engine 1 tore. Engines 1 and 2a are wider still, with worst frames 29.3 and 25.3 dB down. At the 5th percentile, the floor a viewer meets a few times a minute, SmoothMotion is 3.2 dB below a cross-fade, Engine 2b 4.1 dB, and the two early engines 22 dB (Figure 6b, dashed). At the single worst frame the gap is far wider. At 4× the pattern holds but narrows: SmoothMotion's worst frame falls to 13.9 dB below a cross-fade, against 19.1 for Engine 2b.

So the honest statement of the decision is this. I replaced a more accurate engine with a less accurate one because the less accurate one fails more gently. On this footage that cost about 2 dB on average, and bought a worst case about 12 dB shallower at 2×. Whether that is the right trade is a judgement about what a viewer notices more, a frame that is slightly soft all the time or a frame that is badly wrong once in a while; the reputation this essay is about suggests the second. But I made that call on a handful of review clips without these numbers, and the numbers say the cost was larger than I assumed. The obvious next engine keeps SVP's smooth renderer and finds better vectors for it. Section 7.8 tests that idea, and it does not work as well as I expected.

7.4. The 320-pixel cap

The cap was introduced to make a budget phone keep up, and it was never checked for what it costs. The harness can run the same engine with its finest search level at other sizes, so I did. On the desktop GPU the cap reproduces the shape of the Adreno 613 result: with it, the search takes 3.5 to 5.4 ms per frame pair at every size from 480p to 2160p; without it, 13.7 ms at 480p rising to 286 ms at 2160p (Figure 9a). Those are not phone milliseconds; the phone was about 23 times slower at 480p. But the line is flat with the cap and steep without it, which is the claim, and Equation (8) predicts both lines (Table 1).

In quality, the cap sits at the knee (Figure 9b, Table 4). Halving it to 160 px costs 2.7 dB at 2×; doubling it to 640 px changes nothing at 2× and gains 0.4 dB at 4×; searching the full 1080p frame gains 0.4 and 1.1 dB for twenty times the search time. On the smaller Tears of Steel frames, whose long side is 854 px, the cap's effect is under half a decibel in every direction. For ordinary motion the cap costs almost nothing. For motion twice as fast between source frames, there is a measurable gain the phone cannot currently afford.

7.5. What the safety nets cost

Table 4 also switches off, one at a time, the parts of the renderer that exist to make failures gentle. The results are not all flattering. The bad-area mask, which fades poorly matched blocks toward a cross-fade, *lowers* average PSNR on this footage, by 0.8 dB at 2×; it trades accuracy on blocks it distrusts for safety, and here most of those blocks were right. Its benefit at the worst frame is small (4.6 against 5.3 dB below a cross-fade). The coverage mask changes the mean by less than 0.02 dB, because these clips have little of the uncovered background it exists for. The smoothness penalty in the search is worth about 0.1 dB on average and 2.6 dB at the worst frame. And the hard-pair rules rarely ran: of the 168 BBB frame pairs, the scene classifier called 7 hard at 2× and 14 at 4×, none in Tears of Steel, and none a cut, so the re-timing and clamping

Table 3. Rebuilt-frame quality against the real frame. PSNR in dB (higher is better), SSIM on luma, and "worst": the largest margin by which any single rebuilt frame fell below a plain cross-fade of the same pair. Means are over clips. Big Buck Bunny (BBB) is 1080p at 60 fps, 14 segments; Tears of Steel (ToS) is 854×356 at 24 fps, 5 clips. Engine 1 draws midpoints only, so its 4× score covers $t = 0.5$ alone.

	BBB, 2× (30 → 60)			BBB, 4× (15 → 60)			ToS, 2× (12 → 24)		
Method	PSNR	SSIM	worst	PSNR	SSIM	worst	PSNR	SSIM	worst
Repeat frame	36.1	0.932	−16.6	33.2	0.915	−19.8	32.5	0.873	−7.8
Cross-fade	39.9	0.938	—	37.2	0.924	—	36.8	0.913	—
Engine 1 (27 Sep)	31.6	0.923	−29.3	29.7	0.901	−28.7	31.9	0.910	−19.7
Engine 2a, fragment	33.1	0.945	−25.3	32.4	0.932	−22.6	35.4	0.937	−11.2
Engine 2b, per-pixel	43.2	0.993	−16.6	40.5	0.989	−19.1	38.8	0.962	−2.1
SmoothMotion (current)	**41.2**	**0.985**	**−4.6**	**38.5**	**0.972**	**−13.9**	**38.4**	**0.954**	**−1.6**

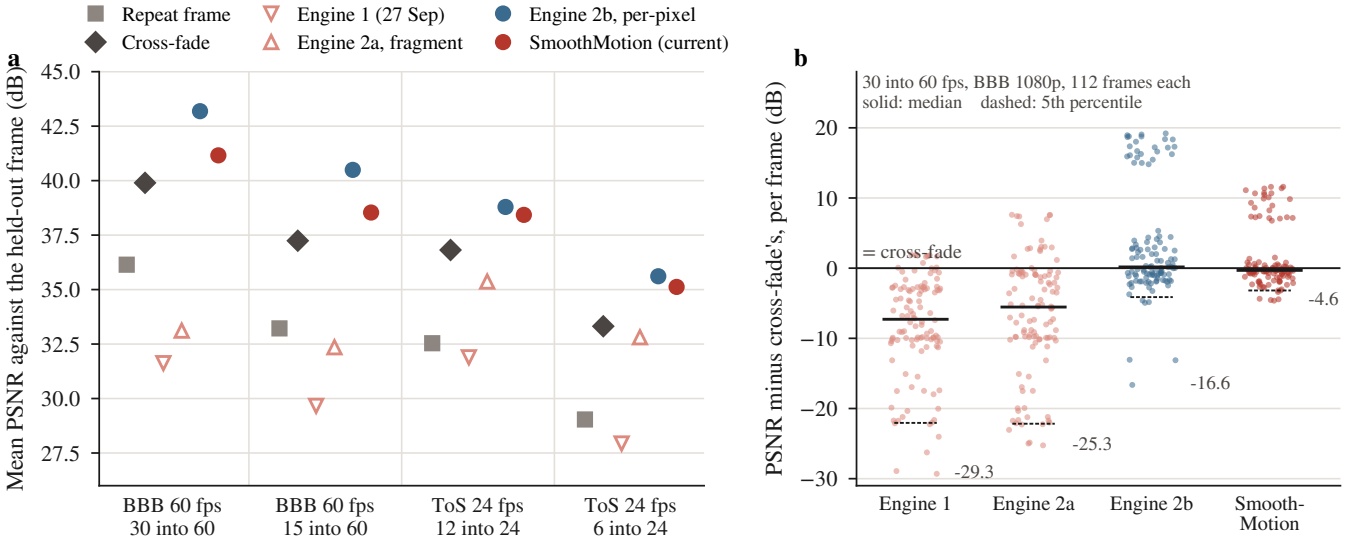


Figure 6. (a) Mean PSNR of rebuilt frames by method and test. (b) Every rebuilt frame of the BBB 2× test, as its PSNR minus the cross-fade's for the same pair; numbers mark each engine's worst frame. Engines 1 and 2a fail loudly and often. Engine 2b is the most accurate on average but has the deepest failures of the two later designs; SmoothMotion gives up some of the top for a much shallower floor.

Table 4. Variants of the current engine on BBB 1080p: mean PSNR (change from as-shipped) and the worst frame relative to a cross-fade.

	2× (30 → 60)		4× (15 → 60)	
Variant	PSNR	worst	PSNR	worst
As shipped	41.16	−4.6	38.54	−13.9
Search at 160 px	38.49 (−2.67)	−10.9	37.68 (−0.86)	−13.9
Search at 640 px	41.09 (−0.07)	−5.4	38.94 (+0.41)	−4.9
Search at full size	41.52 (+0.36)	−13.6	39.66 (+1.13)	−5.0
No bad-area mask	41.99 (+0.83)	−5.3	39.02 (+0.48)	−13.9
No coverage mask	41.16 (+0.00)	−4.6	38.55 (+0.02)	−13.9
No smoothness penalty	41.05 (−0.11)	−7.2	38.45 (−0.09)	−13.9

of hard pairs, and the cut rule, are barely exercised by these two films. Sintel exercises them (Section 7.7).

7.6. Against SVP itself

SmoothMotion's renderer is a port of SVP's, and the second version of this essay could only say that the port matched open-svpflow's output when given open-svpflow's vectors (Table 7). That is a test of the port, not of the design. The fairer question is how SVP's whole method, its own motion search included, does on the same frames. I ran open-svpflow's analysis and frame-rate plugins on every clip, at the same source and output rates, with the scene mode set to draw every tick. Its default mode shows a real frame on some ticks, which this test would score as misses. I tried SmoothMotion's own settings (8×8 blocks, Algorithm 21) and open-svpflow's defaults, and report the defaults, which scored better. SVP works in YUV, so its frames are converted from RGB and back with BT.709 coefficients. That round trip is a small handicap SmoothMotion does not pay, and I have not separated it out.

On the two calmer films, SmoothMotion is ahead of SVP everywhere (Table 5). On Big Buck Bunny the gap is large: 41.2 against 37.4 dB at 2×, with SVP below a plain cross-fade on average. That result is not a few broken frames. SVP's 5th-percentile frame is 16.5 dB below a cross-fade, against SmoothMotion's 3.2, and it falls below a cross-fade on 62% of frames. On Tears of Steel the gap is 0.8 to 0.9 dB. Because the two renderers are the same algorithm, these differences are almost entirely the motion search: SmoothMotion's runs on a contrast-normalised pyramid with a smoothness penalty, SVP's on the raw image with its own block sizes and penalties. I have not traced exactly which difference matters on which

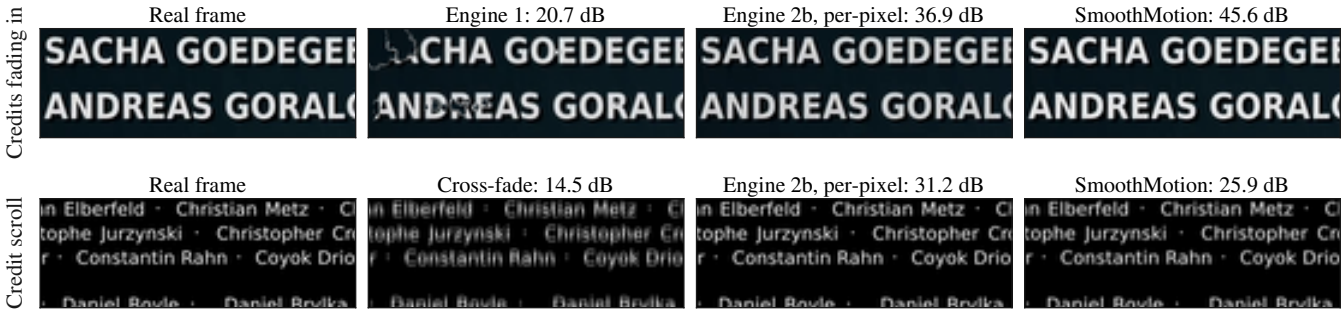


Figure 7. Two rebuilt frames from BBB 2×, cropped from 1080p at full resolution, with each frame's PSNR. Top: credits fading in on a still background. Engine 1 tears the letters; the later engines do not. Bottom: the scrolling end credits, the hardest case for SmoothMotion. A cross-fade draws every line twice. The per-pixel Engine 2b tracks the scroll almost exactly; SmoothMotion's smooth fields leave a soft double edge on the letters. That is the soft failure the design chose, and here it is the less accurate one.

Table 5. Every method on every test: mean PSNR in dB (higher is better) and LPIPS (lower is better), means over clips. Sintel (1920×818, 24 fps, 14 segments) adds eleven scene cuts and fast fights; its PSNR is capped at 60 dB per frame (Section 5.7). SVP is open-svpflow [19]. The hybrids are SmoothMotion's renderer fed Engine 2b's vectors (Section 7.8). Engine 1 draws midpoints only, so its 4× scores cover $t = 0.5$ alone.

	Big Buck Bunny, 60 fps				Tears of Steel, 24 fps				Sintel, 24 fps			
	2×		4×		2×		4×		2×		4×	
Method	PSNR	LPIPS	PSNR	LPIPS	PSNR	LPIPS	PSNR	LPIPS	PSNR	LPIPS	PSNR	LPIPS
Repeat frame	36.1	0.014	33.2	0.022	32.5	0.038	29.0	0.058	17.8	0.274	14.8	0.390
Cross-fade	39.9	0.026	37.2	0.025	36.8	0.069	33.3	0.070	20.3	0.255	17.5	0.320
Engine 1	31.6	0.059	29.7	0.090	31.9	0.093	27.9	0.144	19.3	0.335	15.9	0.438
Engine 2a, fragment	33.1	0.033	32.4	0.036	35.4	0.067	32.8	0.074	20.2	0.292	17.9	0.354
Engine 2b, per-pixel	43.2	0.007	40.5	0.010	38.8	0.042	35.6	0.051	19.2	0.260	17.1	0.316
SVP (open-svpflow)	37.4	0.024	35.2	0.025	37.6	0.060	34.2	0.068	21.5	0.240	18.7	0.312
SmoothMotion	**41.2**	**0.010**	**38.5**	**0.018**	**38.4**	**0.052**	**35.1**	**0.064**	**20.4**	**0.226**	**18.3**	**0.291**
Hybrid, 320 px grid	41.4	0.010	39.1	0.016	38.5	0.051	35.3	0.062	20.8	0.223	18.4	0.292
Hybrid, full-size grid	41.5	0.011	39.6	0.014	38.2	0.048	35.0	0.058	21.1	0.219	18.5	0.291

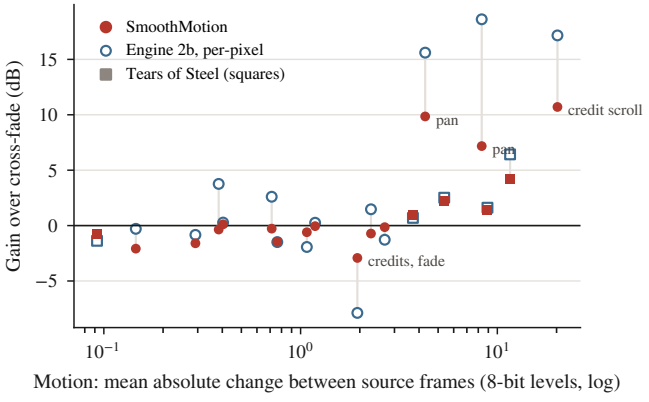


Figure 8. Each clip's mean gain over a cross-fade at 2×, against how much it moves. Filled red: SmoothMotion; open blue: Engine 2b; circles BBB, squares ToS. Below about one grey level of change, interpolation barely matters and both sit near zero. With real motion, both gain several decibels, and the per-pixel engine gains more.

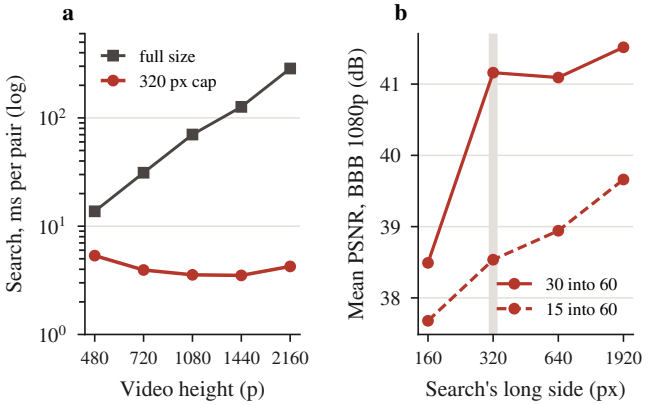


Figure 9. (a) GPU time of the motion search per frame pair on an Intel UHD 630, with the 320 px cap and searching the full frame; median of 10 pairs at each size, after warm-up. (b) Mean PSNR on BBB 1080p as the search's finest level is set to 160, 320 (as shipped), 640 and 1920 px.

footage, so I will only claim the measurement. On this footage, porting SVP's renderer and writing a new search was better than porting both.

Sintel reverses the order on PSNR. There, SVP is the most accurate design at 21.5 dB against SmoothMotion's 20.4 at 2×, and the only one that rarely falls below a cross-fade: 8% of frames outside the fade, against 46% for SmoothMotion. The bottom row of Figure 12 shows the mechanism. When the search cannot follow a fight, SVP's renderer falls back toward a blurred, ghosted average, and an average is exactly what PSNR rewards when the true answer is uncertain. SmoothMotion's cut rule instead shows a sharp real frame from the

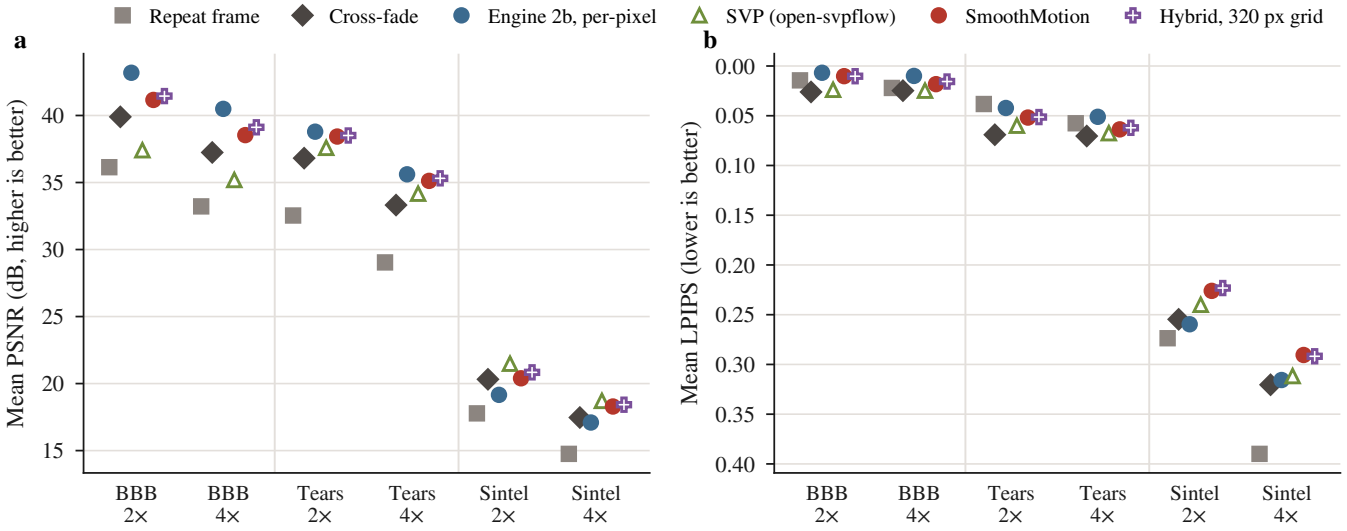


Figure 10. (a) Mean PSNR and (b) mean LPIPS (axis inverted, so higher is better in both panels) for the main designs on all six tests. On the two calmer films the ranking is stable: Engine 2b, then SmoothMotion and the hybrid, then the baselines. SVP falls below a cross-fade on Big Buck Bunny. On Sintel everything collapses toward the baselines, SVP leads on PSNR, and SmoothMotion and the hybrid lead on LPIPS.

wrong moment. The perceptual measure disagrees with PSNR here and ranks SmoothMotion ahead (0.226 against 0.240). Section 7.9 comes back to the disagreement.

One caution about scope. This is open-svpflow, not the commercial SVP application with its GPU paths, tuned profiles and optional neural interpolation [3]. It is the closest open implementation of SVP's method, and the one SmoothMotion's renderer descends from, but it is not a measurement of the product people buy.

7.7. Cuts and chaos

Sintel is the test this engine's caution was built for, and Figure 11 shows every rebuilt frame. Three results stand out.

First, the cut rule works. All eleven pairs that span a cut were classified as cuts, at both 2× and 4×, and drawn as the nearer real frame. On those frames SmoothMotion averages 3.2 dB above a cross-fade at 2×, where Engine 2b, with no cut rule, averages 2.5 dB below one by warping one shot into the next. But "nearer" is ambiguous at the midpoint. At $t = 0.5$ the engine shows the later frame, which is right when the deleted frame falls after the cut and wrong when it falls before it. The top two rows of Figure 12 are one of each: 24.5 dB when the guess is right, 4.9 dB when it is wrong. A viewer would see the wrong row as a cut arriving one drawn frame early, which at 24 → 120 is a single 8 ms tick. That costs the decibels and probably costs nothing to the eye. At 4× the ambiguity only arises at $t = 0.5$, and the cut frames average 0.9 dB above a cross-fade.

Second, the rule fires far too often. Of the 101 pairs at 2× that do not span a cut, the classifier called 38 cuts, and 28 of 45 at 4×, mostly in the fights, where most blocks match badly because everything moves at once. Calling a fight a cut is a safe error: the viewer sees real frames at the source rate, which is what they would have seen with the feature off. It is also why SmoothMotion gains so little over a cross-fade on this footage (0.3 dB on uncut frames, outside the fade). It is not interpolating most of them. The hard-pair rules ran as well, with 43 pairs classed hard. On the frames it does draw, the engine's floor holds. Outside the fade its worst frame is 5.3 dB below a cross-fade, and so is SVP's, against 5.8 for Engine 2b, which falls below a cross-fade on 78% of Sintel frames.

Third, there is a failure I did not anticipate. The worst frame of SmoothMotion, Engine 2b and the hybrids, about 36.9 dB below a cross-fade, is in the flash and fade to black at 7:15. A fade changes every pixel's brightness by the same fraction, so nearly every block matches badly and the classifier calls it a cut. But a fade between two frames is exactly what a cross-fade draws, so the cross-fade is almost perfect (48.5 dB) and the real frame the cut rule shows is far off (11.7 dB). The fix is cheap and specific: before calling a cut, check whether one frame is a uniform brightness scaling of the other. That fix is now on the list. A fade is the most common transition in film after the cut, so this is not an edge case.

7.8. Better vectors for the smooth renderer

Section 7 ended the comparison with Engine 2b on an obvious proposal: keep SVP's smooth renderer and give it better vectors. The harness can test that directly. I saved Engine 2b's final level-0 vector fields for every pair (8 px blocks on a frame up to 1920 px wide, each with a confidence) and fed them to SmoothMotion's renderer in place of its own search. Each SmoothMotion block takes the confidence-weighted mean of the Engine 2b vectors inside it, rescaled to its own pixel size. Everything after the search is unchanged: the field stage, the masks, the scene classifier and the renderer. I ran the hybrid on SmoothMotion's shipped 320 px grid and on a full-size grid, which keeps all of Engine 2b's detail.

It helps, but not much (Table 5, Figure 13). On Big Buck Bunny at 2×, where Engine 2b leads SmoothMotion by

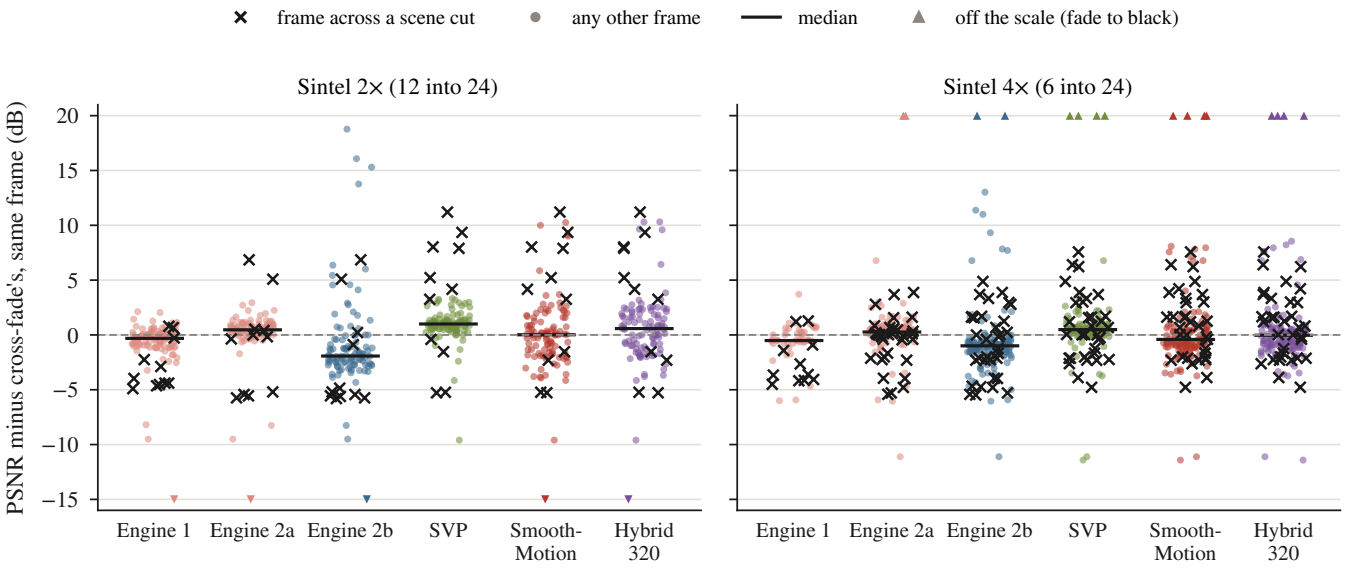


Figure 11. Every rebuilt Sintel frame, as its PSNR minus a cross-fade's for the same pair. Crosses: frames in a pair that spans a scene cut. Bars: medians. Triangles: off the scale, all from the flash and fade to black (segment at 7:15), where every rebuilt frame is either exact or, at a fade, far worse than a cross-fade. Nothing beats a cross-fade by much on this footage. The designs differ mostly in how far below it they fall.



Figure 12. Three Sintel frames rebuilt at 2×, cropped from 1920×818 at full resolution, with each frame's PSNR. Top: the deleted frame is just after a cut. The cut rule shows the later real frame and is right; Engine 2b warps the previous shot. Middle: the deleted frame is just before a cut. The same rule shows the later frame and is wrong; Engine 2b, warping the earlier frame, is right. Bottom: a fight with no cut. Engine 2b's frame is sharp but misplaced, SVP's is a ghosted blur, and SmoothMotion, having called the pair a cut, shows a sharp real frame. On this frame both PSNR and LPIPS rank SVP's blur first; across Sintel, LPIPS more often does not (Section 7.9).

2.0 dB, the hybrid recovers 0.2 dB on the shipped grid and 0.3 dB on the full one. At 4× it recovers 0.6 and 1.1 dB of the 2.0. On Tears of Steel it changes little either way, and on Sintel it gains 0.4 to 0.7 dB at 2×. The worst frames move in both directions: at 4× on Big Buck Bunny the hybrid's worst frame is 4.8 dB below a cross-fade against SmoothMotion's 13.9, but at 2× it is 10.6 against 4.6.

So most of Engine 2b's lead was never in its vectors. It was in drawing: Engine 2b moves every pixel by a vector of its own and settles overlaps pixel by pixel, while SVP's renderer, given the same vectors, blends them smoothly between block centres. That smoothness is precisely what Section 5 identified as failing soft. The renderer cannot make two neighbouring pixels disagree, so it cannot draw the sharp boundary of a moving letter either. Accuracy and safety here are one property seen from two sides, and the obvious next engine would buy back between a tenth and a half of the gap. The more promising design is a renderer that is per-pixel where the vectors are confident and smooth where they are not. That is a renderer project, not a search project.

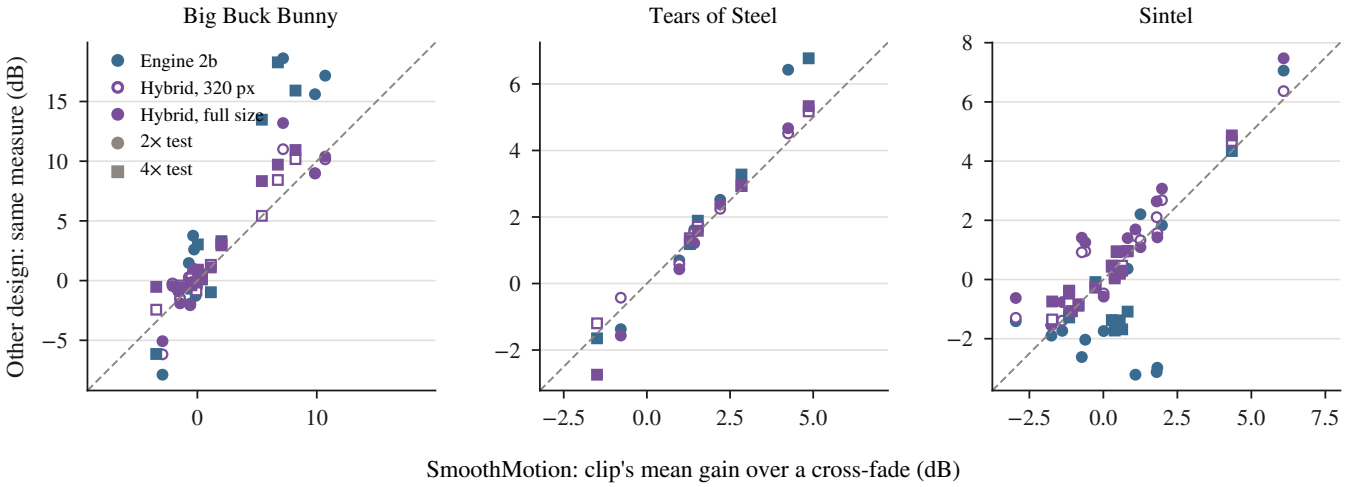


Figure 13. Each clip's mean gain over a cross-fade: SmoothMotion on the horizontal axis, the other design on the vertical. Points above the diagonal beat SmoothMotion. Engine 2b (blue) pulls far ahead on Big Buck Bunny's pans and credits. The hybrids, which give SmoothMotion's renderer Engine 2b's vectors, stay close to the diagonal. Sintel's fade segment is left out (Fig. 11).

7.9. A perceptual measure

PSNR and SSIM measure distance from the real frame, and the second version of this essay warned that they are the wrong tool for deciding what a viewer would watch. LPIPS (Equation 19) is the standard learned alternative, fitted to human judgements of image similarity [35]. It is still not a viewer watching motion, but it disagrees with PSNR in instructive ways.

On Big Buck Bunny the two agree almost completely. On 99% of the frames where SmoothMotion and Engine 2b differ measurably, both measures pick the same winner, and LPIPS keeps the PSNR ranking at the top: Engine 2b, then SmoothMotion and the hybrids. One change of order matters. By LPIPS, a cross-fade is *worse* than repeating the frame on both calmer films (0.026 against 0.014 on Big Buck Bunny at 2×). The network judges a double image as further from the truth than a sharp frame from the wrong moment, while PSNR judges the opposite. On Tears of Steel, whose 854 px frames move only a few pixels between frames, LPIPS even prefers repeating the frame to SmoothMotion on 70% of the frames where they differ, while PSNR prefers SmoothMotion on 96%. Any interpolation softens the frame slightly, and on small, slow motion the softening costs more, by this measure, than the motion it recovers.

On Sintel the two measures stop agreeing. Comparing SmoothMotion with SVP frame by frame, they pick the same winner only 48% of the time, a coin toss. PSNR prefers SVP on 76% of frames and LPIPS prefers SmoothMotion on 73%. The likely reason is the one in Section 7.6: when the motion is too hard to follow, the minimum-error guess is a blur, and a perceptual measure punishes blur more than PSNR does. It does not always win. On the fight frame in Figure 12, LPIPS also prefers SVP's blur. By LPIPS, SmoothMotion and its hybrids are the best designs on Sintel at both rates, and Engine 2b, the most accurate design on the calm films, is behind a cross-fade at 2×.

Table 6. When SmoothMotion stands down, and why. All of these pass frames through untouched at the cost of one texture copy [7, 26].

Condition	Reason
Off in settings (the default)	It is the viewer's choice, made through a warning
Live stream	Not attempted
HDR video	GL tone mapping differs by device
Thermal status moderate or worse	It costs heat; the phone comes first
Battery saver on	The user has already said what matters
Source within 15% of the screen	Nothing to gain
Over 10% of outputs dropped for ~10 s	The phone cannot keep up with this video
Screen held below the output for 5 s	Output drops to what the screen shows
No OpenGL ES 3.2	The setting is shown disabled, with the reason
Any engine failure	Plain playback for that player

None of this settles which engine a person would prefer to watch; that needs people. What it does show is that the engines make different kinds of error, and that a single number hides the kind. A design that is safe by PSNR blurs; a design that is safe by LPIPS shows real frames. SmoothMotion's caution is of the second kind, which is also the kind a viewer has already accepted every time they watched a film without interpolation.

8. Knowing When to Stop

If I had to summarise the engineering in one sentence, it would be that the engine's most important output is "no".

Every half second a governor on the player's thread decides whether interpolation should run and at what rate, and the conditions under which it says no outnumber the conditions under which it says yes. Table 6 lists them.

Two things about this list matter for the argument.

First, every condition is visible. The status the governor

publishes is shown in the player's settings in both apps: "24 → 120 fps" when it is working, and "Paused: battery saver" or "Not needed: already 60 fps" when it is not [7]. Television interpolation never told anyone it was running. This tells you when it is not.

Second, the failure policy is asymmetric on purpose. The worst outcome for a viewer who turned the feature on is not that it stops; it is that it makes their video worse. A phone that drops frames gets plain playback for that video at that quality, and the next video gets a fresh chance. A GPU whose driver rejects a shader gets plain playback, not an error. Playback is never the thing that breaks.

The device log in Figure 5 adds a third point, against myself. Every gate in Table 6 watches a quantity the governor can count. The cadence bug broke none of them. A system whose honesty rests on counters needs at least one counter for the thing the viewer actually sees, and the number of untouched decoded frames per second, which the log already makes possible to compute, is the obvious one to add.

9. WHAT IT COSTS, AND WHAT I DON'T KNOW

The first essay complained that energy figures for a consumer product had to be recovered by outside researchers because the vendor had not published them [9]. I am not going to commit the same offence with my own software, so here is the ledger, updated, with the empty rows left in.

The empty rows are still the important ones. I do not know how much battery SmoothMotion costs. The library's documentation says it uses more battery and makes the phone warmer, which is certainly true and tells you nothing about how much [7]. Both apps keep it off by default for that reason, and the governor's thermal and battery-saver gates exist because the cost is real, but a gate is not a measurement. The test is simple and needs a connected phone: the same video for half an hour with the feature on and off, reading the battery's charge counter, ideally on more than one GPU.

Four rows moved from pending to measured, and the measurements changed my account of my own engine twice: once when the engine I replaced turned out more accurate, and again when better vectors turned out not to be the reason. That is what measurements are for.

Table 7. The evidence, and its basis. "Pending" means not measured.

Claim	Basis
Holds 1080p 23.98 → 120 fps	Mali-G615 MC6, 78,361 frames emitted, 0.44% dropped; display rose 60 → 120 Hz on its own [28] (Fig. 5)
Full rate on a budget GPU	Adreno 613, 360p and 480p, 30 → 60 fps [7]
Search ~23 ms per pair	Adreno 613, developer measurement [18]; flat-cost shape reproduced on a desktop GPU (Fig. 9)
Renderer matches open-svpflow	43.2–44.7 dB against a 46.7 dB ceiling, given SVP's vectors [23, 17]
Better than repeat and cross-fade	**Measured**: 610 held-out frames, three films, four engines; on calm footage, not on fights (Sec. 7)
Fails softer than earlier engines	**Measured**, at a cost of about 2 dB on average (Sec. 7)
Comparison with SVP itself	**Measured** against open-svpflow: ahead on two films, behind on Sintel by PSNR, ahead by LPIPS (Sec. 7.6)
Footage with cuts and chaos	**Measured**: every cut caught; a third of other hard pairs also called cuts; fades mistaken for cuts (Sec. 7.7)
Better vectors would close the gap	**Measured**: no, 0.2–1.1 dB of 2.0 (Sec. 7.8)
Cadence fix	In code; not confirmed on a device [29, 30]
Battery cost	Pending
Thermal behaviour over time	Pending
Devices tested	Two GPUs on phones; one desktop GPU offline

10. DISCUSSION

10.1. Why not a neural network

The best-known interpolation methods in research are now learned: RIFE, published at ECCV in 2022, estimates intermediate flow with a network and runs several times faster than the learned methods before it [37]. SVP itself now offers RIFE as an option on desktop GPUs [3]. A phone's budget is different. The engine has to produce up to four frames per decoded frame within about 42 ms on a mid-range GPU, alongside the video decoder and the app's interface, without dedicated neural hardware being guaranteed and without shipping model weights in a music player. Block matching on a 320 px pyramid plus SVP's renderer fits that budget on the hardware I have measured. A learned model on a phone's NPU may well do better, and Section 7 now gives it a harness and a number to beat.

10.2. Video is the easy case

Interpolation in video players is often discussed alongside frame generation in games, and the second engine was written after studying the game version [21, 22]. They are different problems, and the difference favours video. A game has to show a generated frame before the next real frame exists, so it predicts, and the prediction adds latency between the player's input and the screen. A video player already has the next frame: the decoder runs ahead of the display, and Media3 holds each output until its presentation time [7]. Every frame SmoothMotion draws is between two frames it already has. Nothing is predicted, and nothing anyone does with their hands is delayed.

That is also why the engine can afford to be cautious. It can look at both frames before deciding whether a pair is a cut, a hard pair or an easy one, and it can show the real frame instead of a guess whenever the guess would be bad.

10.3. Who decides

The television story ended with an industry standard for turning the feature off. That was the right outcome for televisions, where the feature was imposed. On a phone, inside an app, with the switch beside the video and a status line that says what is happening, the decision is where it belongs: with the person watching, on that video, at that moment. Someone watching a football match on a bus and someone watching a film they love are entitled to different answers, and the software's job is to make both easy and to be honest about which one is running.

10.4. Limitations

This essay is written by the engine's author about the engine, and its evidence is mostly the author's own. The quality measurements run the shipped shaders on a desktop GPU, not a phone, on 610 frames from three open films, two of them animated. Fourteen hand-picked Sintel segments are a small sample of cuts and fights, and say nothing about what viewers prefer. PSNR and SSIM measure distance from a real frame, which is not the same as visible quality, and they rank Engine 2b above the engine I chose. LPIPS is closer to perception but was fitted to still images, not motion. The SVP comparison is with open-svpflow, not the commercial application. The device evidence is two GPUs and one detailed log. The search timings on phones are developer measurements reported in commit messages. The battery and thermal costs are unmeasured. The cadence fix is unconfirmed on a device. The engine depends on Media3 1.11.0 behaviour that is not public API, read from its bytecode, and a Media3 release could change it [38]. And the central argument, that television interpolation's reputation is a property of deployment, is partly an argument about taste, which no measurement settles. I have tried to keep the two apart.

11. CONCLUSION

Here is the receipt, with the current numbers. A 24 fps film on a 120 Hz phone shows each frame five times. SmoothMotion replaces four of those five with frames drawn on the phone's GPU, keeps the fifth untouched, and lets the display raise itself to 120 Hz by emitting a steady clock. Its search runs on a frame no wider than 320 pixels, which took a budget GPU from about 310 ms to about 23 ms per pair [18] and, measured here, costs almost nothing in quality at ordinary motion. Against held-out frames on ordinary footage its drawn frames beat repeating the frame and beat a cross-fade, and at their worst fall a few decibels short of a cross-fade, where the first engine I wrote fell nearly thirty. One phone held 1080p at 120 fps for twelve minutes with under half a percent of frames dropped, and the same log shows a cadence bug that no counter caught. It stands down for heat, battery saver, HDR, live video and any phone that cannot keep up, and it says so on screen.

The measurements also show the price of the design. The engine I replaced was about 2 dB more accurate on average, and giving the new renderer its vectors recovers only part of that, because the accuracy and the safety live in the same place. Against SVP's own method on the same frames, SmoothMotion is ahead on calm footage. On fights it trades PSNR for sharpness and wins on the perceptual measure. Every cut in the test was caught, and fades, which the engine mistakes for cuts, are the next fix. I chose a softer, safer engine over a sharper, riskier one, and the evidence now supports the safety and puts a number on what it costs. What the engine still lacks is a battery figure, a viewer study, and a confirmed fix for the one bug a user found. Those are the next tests.

Televisions taught people that interpolation is something done to them. It can be something they ask for. The frames were never shot, and drawing them is not a betrayal of the film. Drawing them without asking was.

A. REPRODUCING THE MEASUREMENTS

The scripts behind Section 7 and Figures 5 to 9 accompany this essay in its `analysis` folder. `engine.py` runs the current `ComputeMotionEngine` from its Kotlin source with moderngl; `experiment.py` produces every SmoothMotion, variant and baseline row; the earlier engines are run with the harnesses written for them at the time, from the files at commits `71a2b73` and `c333341`; `score_old.py` scores their output; `timing.py` produces Figure 9a; `devicelog.py` turns the bug report's log into Figure 5; and `figs_*.py` draw the figures and tables. Big Buck Bunny segments are 17 frames from `bbb_sunflower_1080p_60fps_normal.mp4` starting at 45, 105, 135, 170, 205, 240, 280, 310, 390, 430, 470, 495, 515 and 545 s; the Tears of Steel clips are the five 10-frame extracts used during development, at 854×356. The full result set is 2,970 rows for the current engine and baselines and 858 for the earlier engines, one per rebuilt frame.

The third version's tests are in the same folder. `sintel_scan.py` chose the Sintel segments from the 1080p release (`Sintel.2010.1080p.mkv`, 24 fps), starting at frames 1009, 1361, 4320, 5964, 6212, 6891, 7342, 7712, 7962, 9328, 10442, 13032, 13424 and 14100, and `sintel_cuts.json` holds the hand-checked cut labels. `run_sintel.sh` runs the earlier engines and open-svpflow on them (`svp_run.py`, through VapourSynth), and `harness2b_dump.py` saves Engine 2b's vector fields for the hybrid. `experiment2.py` renders SmoothMotion, both hybrids and the baselines, and scores every method on all three films by PSNR, SSIM and LPIPS (the `lpips` package, AlexNet, version 0.1); `run_exp2_parallel.sh` runs it as four workers. `figs_v3.py` draws Figures 10 to 13 and Table 5. That set is 5,246 rows.

REFERENCES

[1] MUBI Notebook, "Dreams Per Second: A History of the Frame Rate." `https://mubi.com/notebook/posts/dreams-per-second-a-history-of-the-frame-rate`

[2] Wikipedia, "Display motion blur" (motion interpolation and the soap opera effect), accessed November 2026.

https://en.wikipedia.org/wiki/Display_motion_blur

[3] SmoothVideo Project. https://www.svp-team.com/

[4] ITV News, "Tom Cruise's new mission: To get people to watch movies properly on their modern TVs," 5 December 2018. https://www.itv.com/news/2018-12-05/tom-cruises-new-mission-to-get-people-to-watch-movies-properly-on-their-modern-tvs

[5] Engadget, "LG, Panasonic and Vizio fight motion smoothing with 'filmmaker mode'," 27 August 2019. https://www.engadget.com/2019-08-27-motion-smoothing-4k-filmmaker-mode.html

[6] Koda for Android, source and history. https://github.com/Ivorisnoob/Koda

[7] SmoothMotion (smoothmotion-android) 0.1.0: README, docs/ARCHITECTURE.md and docs/TRAPS.md. https://github.com/Ivorisnoob/smoothmotion-android

[8] OpenStream 2.2 release notes, 10 October 2026. https://github.com/Ivorisnoob/OpenStream/releases/tag/v2.2

[9] Ivor, "Breathing Something That Could Have Been Air: On the Opportunity Cost of Generative Compute," essay, 2026.

[10] Android Developers, "Frame rate." https://developer.android.com/media/optimize/performance/frame-rate

[11] Android Central, "OnePlus aims for max smoothness via chip capable of 30fps to 120fps upgrade," 2020. https://www.androidcentral.com/oneplus-aims-max-smoothness-chip-capable-30fps-120fps-upgrade

[12] Pixelworks, "Pixelworks Powers OnePlus 11 Smartphone to Redefine Visual Excellence for Mobile Gaming," press release, January 2023. https://technode.global/prnasia/pixelworks-powers-oneplus-11-smartphone-to-redefine-visual-excellence-for-mobile-gaming/

[13] Pixelworks, "Pixelworks Partners with Xiaomi to Redefine the Excellence of Mobile Gaming Experience," press release, August 2023. https://technode.global/prnasia/pixelworks-partners-with-xiaomi-to-redefine-the-excellence-of-mobile-gaming-experience/

[14] Gizmodo, "The Trouble With LCD TVs: Motion Blur and the 120Hz Solution." https://gizmodo.com/the-trouble-with-lcd-tvs-motion-blur-and-the-120hz-sol-290237

[15] Salon, "Why does 'The Hobbit' look so weird?" 6 December 2012. https://www.salon.com/2012/12/06/why_does_the_hobbit_look_so_weird/

[16] Koda commit 71a2b734, "Add opt-in Smooth motion video frame interpolation," 27 September 2026.

[17] ComputeMotionEngine.kt, smoothmotion-core, and THIRD_PARTY_NOTICES.md, in [7].

[18] Koda commit 6e62131c, "Find Smooth motion's motion on a small frame so budget phones keep up," 28 September 2026.

[19] Z1xus, open-svpflow, "An SVPFlow reimplementation," Apache-2.0; ported from commit a334f02. https://github.com/Z1xus/open-svpflow

[20] SmoothVideo Project, SVPflow plugins; the method open-svpflow reimplements. See [3].

[21] GamingOnLinux, "AMD FidelityFX Super Resolution 3 (FSR3) is now open source," December 2023. https://www.gamingonlinux.com/2023/12/amd-fidelityfx-super-resolution-3-fsr3-is-now-open-source/

[22] Koda commit c3333414, "Smooth motion up to 120 fps with a new compute engine," 28 September 2026.

[23] Koda commit 6e9e30e2, "Rebuild Smooth motion on SVP's frame interpolation method," 28 September 2026.

[24] Koda commit 8df20417, "Draw every Smooth motion frame and cut its GPU cost," 28 September 2026.

[25] Koda commit 11a46a3e, "Stop Smooth motion switching itself off by mistake, and show its status," 28 September 2026.

[26] Koda commit 0309e313, "Add a Smooth motion switch to the video player and give it more time," 28 September 2026.

[27] [22], and "Dropped frames are counted against output, not decoded frames," docs/TRAPS.md in [7].

[28] Koda 5.0 in-app bug report with FrameInterpolation log, Xiaomi 2311DRK48I (Mali-G615 MC6, Android 16), 29 September 2026, 19:39–19:57 IST; shared by the user who filed it.

[29] Koda commit 4bbd6f3e, "Keep Smooth motion's cadence locked after dropped frames," 30 September 2026.

[30] Koda ROADMAP.md, "Smooth motion loses its cadence after a burst of dropped frames," accessed November 2026.

[31] Koda 5.1 release notes, 1 October 2026. https://github.com/Ivorisnoob/Koda/releases/tag/5.1

[32] Blender Foundation, *Big Buck Bunny* (2008), 1080p 60 fps release, CC BY 3.0. https://peach.blender.org/

[33] Blender Foundation, *Tears of Steel* (2012), CC BY 3.0. https://mango.blender.org/

[34] Blender Foundation, *Sintel* (2010), 1080p release, CC BY 3.0. https://durian.blender.org/

[35] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, O. Wang, "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric," CVPR 2018, pp. 586–595. arXiv:1801.03924.

[36] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing* 13(4), 600–612, 2004.

[37] Z. Huang, T. Zhang, W. Heng, B. Shi, S. Zhou, "Real-Time Intermediate Flow Estimation for Video Frame Interpolation," ECCV 2022, LNCS 13674, pp. 624–642. arXiv:2011.06294.

[38] AndroidX Media3 1.11.0; behaviour relied on is listed under "Media3 version" in docs/TRAPS.md, [7]. https://github.com/androidx/media