ChatGPT

SDL3 TTF Font Handling in WORR-2

Background on Font Metrics and SDL_ttf

In TrueType fonts, each glyph has metrics defined relative to a baseline: the font **ascent** (distance from baseline to the highest outline point) and **descent** (distance from baseline to the lowest point) define its vertical extents [1]. An additional **line gap** is typically included to determine the default line spacing (baseline-to-baseline distance) as `line_space = ascent - descent + linegap` [2]. SDL_ttf's `TTF_GetFontLineSkip` returns this recommended line spacing, which includes the font's internal leading (line gap). Each glyph also has **bearing** metrics: for horizontal text, the *top bearing* (often called `bearingY`) is the vertical distance from the baseline to the glyph's top pixel row [3]. The bearingY (or `maxy` in SDL_ttf) tells how far above the baseline a glyph extends. Likewise, `miny` can indicate how far below the baseline the glyph's lowest pixel sits (often negative for descenders). Correct text rendering must account for these metrics: when drawing a glyph at a given baseline position, one should offset it by the bearing values so that it "rests" on the baseline. In fact, the standard formula using FreeType metrics is:

`draw_x = pen_x + bitmap_left` and `draw_y = baseline_y - bitmap_top` [4],

where `bitmap_top` is analogous to bearingY and represents the pixel distance from baseline to the glyph's top. SDL_ttf internally uses similar logic when rendering glyphs to surfaces.

Current Implementation in WORR-2

WORR-2 uses SDL3_ttf (version 3.x) for TrueType font rendering, integrated with a custom glyph atlas system. The font handling code is primarily in `src/client/font.cpp`, and it manages three font types: legacy bitmap fonts, "kfont" bitmap fonts, and TTF vector fonts. For TTF fonts, the code loads the font with a specified **virtual line height** (the intended height in the game's virtual 640×480 coordinate space) and a **pixel scale** factor. Upon loading a TTF, the code queries the font's metrics via SDL_ttf: `font->ttf.ascent = TTF_GetFontAscent()`, `font->ttf.line_skip = TTF_GetFontLineSkip()` [5]. It then computes its own baseline and extent values in a function `font_compute_ttf_metrics()`. This function adjusts for SDL_ttf's behavior by disregarding the line skip if it includes extra gap: it sets `font->ttf.baseline = (ascent > 0 ? ascent : line_skip)` and `extent = ascent + |descent|` [6]. In other words, WORR-2 initially uses the font's ascent as the baseline distance (if available), and ensures the total height (extent) covers ascent plus descent. This was done to remove the font's internal line gap from the effective height, since the engine manages line spacing separately.

Baseline alignment improvements: Because of past issues, the implementation has evolved to compute more precise baseline metrics. WORR-2 preloads all ASCII glyphs (codepoints 32–126) into the atlas on font load, then calculates the actual pixel extents of those glyphs. The highest pixel above the baseline (max_y) among ASCII and the lowest pixel (min_y) below baseline are found. These are used to set a **rendered baseline and extent**. For example, if the font's ASCII characters have a top pixel 7 units above baseline and a bottom pixel 2 units below, the engine will set `rendered_ascent = 7` and `rendered_descent = 2`, so `rendered_extent = 9`. This process replaces reliance on SDL_ttf's lineSkip. The change was introduced to fix an earlier problem: *"TTF text was consistently offset vertically"* when drawn alongside the old 8px console font [7]. The cause was that using `line_skip` (which included line gap) added extra empty space, and using the raw ascent for baseline meant most glyphs (which don't reach the maximum ascent) sat too low [8]. By computing glyph extents from actual rendered bitmaps, WORR-2 ensures the **virtual line height** is filled more tightly. The code now prefers these **rendered metrics** when available: by default `cl_font_ttf_metric_mode = 1`, meaning it will use the measured bitmap ascent/descent instead of the font's outline metrics [9].

After loading, each TTF font in WORR-2 has: `font->virtual_line_height` (the design height in game units, e.g. 8 for console font), and `font->ttf.baseline` and `font->ttf.extent` (or their rendered counterparts) set. It computes a scaling factor `font->unit_scale` such that `virtual_line_height * unit_scale = extent` [10] [11]. This unit_scale ties the font's own size to the engine's 2D coordinate system. All glyph drawing uses this scaling: for a given draw call with scale=1 (normal size), the engine multiplies by `unit_scale` and also a global 1.5× boost factor (more on that below) to get the final pixel scale [12]. The result is that the font's `rendered_extent` maps to the desired virtual line height in the game.

HarfBuzz integration: WORR-2 supports advanced text layout via HarfBuzz (enabled with `USE_HARFBUZZ`). When shaping is used (for proportional and non-fixed-width fonts), the code creates a HarfBuzz font from the TTF data and obtains glyph indices and positioning for each grapheme in the string [13]. Critically, WORR-2 uses HarfBuzz for complex scripts and ligatures but still leverages SDL_ttf for metrics and rasterization. The implementation was revised so that the HarfBuzz output and SDL_ttf metrics stay in sync: it builds a `TTF_Text` layout (SDL3_ttf's new layout API) or manually maps HarfBuzz glyph indices to SDL_ttf glyphs, then uses SDL_ttf's **glyph metrics** for placement [14]. Essentially, after HarfBuzz provides the sequence of glyphs and their advances, WORR-2 looks up each glyph's cached metrics from SDL_ttf (or calls `TTF_GetGlyphMetrics` if not cached) to get bearingX, bearingY, etc., ensuring that the vertical alignment (baseline) uses the same values that were used when the glyph was rendered into the atlas. This strategy was implemented to prevent per-glyph vertical drift. An internal note summarizes: using SDL_ttf's placement info *"keeps SDL3_ttf's baseline/ kerning decisions aligned with the glyph bitmaps and prevents per-glyph top-alignment drift."* [15] In practice, WORR-2 sets `bearing_y` (the top bearing) from SDL_ttf and uses it for all glyphs on a line, rather than aligning each glyph's top separately. This guarantees a uniform baseline.

Virtual screen scaling: All 2D drawing in WORR-2 goes through a *virtual screen* abstraction. The game's UI is designed for a 640×480 coordinate space, which is scaled up to the actual window size with integer scaling to preserve pixel alignment [16]. The renderer sets up an orthographic projection such that (0,0) in virtual units corresponds to the top-left of the drawable area and (virtual_width, virtual_height) to the bottom-right [17]. Depending on the window's aspect ratio, `virtual_width` or `virtual_height` may be smaller than 640 or 480 (whichever dimension doesn't fit evenly), resulting in letterboxing. For example, at 1920×1080, the base scale `s = floor(max(1920/640, 1080/480)) = 2`, yielding a virtual space of 640×540 (which is then cropped to 480 in height with black bars) [18] [19]. All font coordinates (like position and size) are given in this virtual space. WORR-2's font system is aware of this scaling: it computes `pixels_per_unit = s` (for scale=1 draws) [20], and by using integer `s` the fonts stay pixel-perfect. Additionally, the engine allows a `ui_scale` or `con_scale` CVAR that multiplies the UI elements; this ultimately multiplies the `draw_scale` passed to Font drawing functions. The combination of *virtual screen scale* and *draw scale* is handled in `font_draw_scale()`, which produces the actual GL texture scaling for the glyph. Notably, there is a constant `k_font_scale_boost = 1.5` used in that calculation [12]. This means WORR-2 deliberately renders TTF fonts at 150% of the nominal size defined by `virtual_line_height`. For instance, the console font has `virtual_line_height = 8`, but the line height in pixels comes out to about 12. This boost was likely chosen to improve legibility or match legacy visuals. The important effect is that the **baseline**

and spacing calculations also incorporate this 1.5 factor. The function `Font_LineHeight()` returns `ceil(virtual_line_height * 1.5 * scale)` [21], ensuring the engine considers a taller line. All metric scaling (for positioning glyphs) uses the same factor, so the baseline offset and glyph advances are scaled up consistently.

In summary, WORR-2's current TTF implementation loads fonts at a given pixel size (often equal to the virtual height, possibly adjusted by a supersample factor), then scales everything by ~1.5× at draw time. It calculates its own baseline and line height from actual glyph data to avoid SDL_ttf's built-in padding. It integrates HarfBuzz for horizontal spacing (kerning, ligatures) but still relies on SDL_ttf's vertical metrics (bearing and ascent) to place glyphs on the baseline uniformly.

Identified Baseline Alignment Issues

Despite the improvements, some baseline alignment issues persist in WORR-2. The phrase "font alignment is completely wrong" likely refers to text not lining up as expected with other UI elements or having inconsistent vertical positioning. Below we analyze the causes and manifestations of these issues:

- **Line-gap inclusion and top padding:** Initially, using `TTF_GetFontLineSkip` directly caused extra empty space above each line. The lineSkip includes the font's line gap (external leading). In an 8-pixel-high console font scenario, this was disastrous: it introduced padding where none existed in the original bitmap font, pushing text downward. WORR-2's documentation notes that *"TTF scaling used the font's* `line_skip`*, which includes the font's line-gap. This introduced extra top padding in the 8px console line height."* [8] In effect, the text was not vertically centered in the console line box. The team mitigated this by using the font's ascent+descent (extent) as the scaling reference instead of the full lineSkip [22]. However, if any residual line gap remains in the calculations (for instance, if `line_skip` was used as a fallback for baseline in some cases), it could still skew the alignment. The fix ensures the virtual line height corresponds to the font's tight bounding box (extent) rather than its recommended linespace, eliminating intentional top/bottom padding.

- **Baseline calculated from font ascent vs. actual glyph top:** Even after removing line gap, another issue is how the baseline position is chosen. If the baseline offset equals the font's maximum ascent, characters with shorter ascent (e.g. the letter "a" might not reach the top ascent value) will appear lower within the line box. WORR-2 originally did exactly this – it used the font's ascent for baseline placement – and found that *"glyphs that do not reach the full ascent (most glyphs) appeared lower than expected."* [23] The result was a noticeable baseline "sag" for typical lowercase text. The current solution computes a custom baseline from glyph data: specifically, the highest top pixel (max_y) among ASCII glyphs is used as the new baseline height [22]. For example, if none of the ASCII characters fully reach the font's ascent line, the computed `font->ttf.baseline` will be smaller than the TTF ascent. This brings the average text line up. In code, after loading glyphs, they do: `font->ttf.baseline = max(ascent, computed_maxy)` [24] [25], then use that for placing glyphs. This has largely fixed the relative alignment of characters to each other and to the console prompt. All ASCII glyphs now share a common baseline that aligns with the legacy 8x8 console cell: *"TTF glyphs align with legacy console prompt/cursor"* after this fix [26].

However, a potential new issue arises when text contains characters **outside** the ASCII set. The baseline calculation initially only considered ASCII 32–126. If a glyph like "Å" (Latin capital A with ring, U+00C5) or a taller diacritic appears, its top might lie above the current baseline (i.e., `glyph_maxy` for that character is larger). WORR-2 addresses this by updating the rendered metrics on the fly: when any newly rendered glyph has a higher bearingY than the current `rendered_ascent`, it increases the stored `rendered_ascent` (and extent) [27]. In other words, the baseline metric can grow if needed. This ensures such glyphs won't be cut off, but it means the baseline reference for the font can shift during runtime. If this happens mid-game, one line of text might suddenly render slightly higher than previous lines (because the baseline offset increased). This dynamic adjustment might be perceived as an alignment bug if not handled smoothly. It's a rare case – typically one would preload all needed glyphs – but it's worth noting in the context of "persisting baseline issues." The ideal solution is to preload any known high glyphs (for languages or symbols in use) or accept a one-time baseline shift when they first appear.

- **Baseline relative to line height (drift between lines):** Another subtle issue is ensuring that each new line of text starts at a consistent vertical offset from the previous line. Even if single-line text aligns, multi-line layouts could drift if line spacing and baseline offset aren't in harmony. WORR-2 recognized that using the raw metrics could cause slight mismatches between the drawn line height and the spacing used for baseline. They solved this by deriving the baseline position from the **line height ratio** instead of absolute pixels. The formula implemented: `baseline_y = y + line_height * (baseline / extent)` [28]. This ties the baseline directly to the engine's chosen line height. In practice, `Font_LineHeight` returns a rounded value (e.g. 12 px for console), and if the font's baseline is 7 and extent 9 in those units, `baseline/extent ≈ 0.777…`, so baseline_y will be ~0.777 of the line height from the top of the line. By using the same ratio for every line, each line's baseline aligns perfectly on the grid. The WORR-2 console update confirms: *"baseline placement now derives from the line-height ratio… removing the previous drift/top-alignment artifacts."* [28] In code, we see this in the drawing function: they compute `line_height = Font_LineHeight()` and then `baseline_offset = font_ttf_metric_baseline(font) * glyph_scale`, finally `baseline_y = line_y + baseline_offset` [29] [30]. Because `glyph_scale` already incorporates the line height and scale, this is equivalent to the formula above. The result is that the baseline is a fixed fraction of each line's total height. If this ratio were off by even a small amount, you'd notice cumulative error (e.g., lines of text creeping up or down), but since it's linked, multi-line text now stacks correctly. Any remaining baseline issue would then be a constant offset (all lines equally off), rather than a per-line drift.

- **Visual verification of baseline (debug overlay):** WORR-2 introduced extensive debug tools to diagnose baseline and spacing. Using `cl_font_debug_draw 1` draws guide lines: a cyan line for the top of text, a green line for the baseline, and a red line for the bottom (baseline + extent) [31] [32]. If the font alignment is "completely wrong," these guides would show it clearly – for instance, the green baseline line might not intersect glyphs at the correct point. Ideally, the green line should cross through the base of characters like 'x' or the underline of '_', and the red line should just kiss the descenders (like the tail of 'g'). A persistent issue could be that the baseline (green) is offset too high or low. If too high, letters seem to hang below the baseline guide (cursor underscores might appear above where they should be). If too low, there's extra space above characters. Given the fixes in place, a likely culprit for any remaining baseline offset is a mis-match between the chosen baseline metric and the art assets or expectations. For example, maybe the console cursor (a legacy graphic) is drawn at y=baseline+1, expecting an older alignment. If WORR-2's new baseline is off by a pixel, that cursor will not sit exactly under the text. This kind of off-by-one alignment error can make the font look "completely wrong" in context even if it's technically consistent.

In summary, WORR-2 has largely solved per-glyph baseline inconsistencies by using a uniform baseline derived from real glyph data, and tied it to the engine's line spacing to avoid drift. Remaining issues might include dynamic changes when new glyphs extend the metrics, and absolute alignment of the baseline relative to legacy elements or the desired visual baseline (which could be off by a small constant). These will be addressed by further fine-tuning the baseline offset.

Identified Spacing Issues

The term "spacing" issues here refers to horizontal spacing between characters and the overall width/position of text. WORR-2's font code handles spacing through a combination of HarfBuzz (for natural font kerning and layout) and manual adjustments (letter spacing and monospace behavior). Several points of concern and improvement have been identified:

- **Kerning and Ligature Handling:** With HarfBuzz enabled for proportional fonts, the expectation is that the spacing between characters (including kerning pairs like "VA") is handled by HarfBuzz's shaping. WORR-2 correctly uses HarfBuzz's output: it obtains each glyph's advance (`positions[i].x_advance`) and x-offset (`x_offset`) [33] [34] . It then increments the pen position by the advance for each glyph. Importantly, when HarfBuzz is used, SDL_ttf's own kerning is **not** applied on top – the code bypasses `TTF_GetGlyphKerning` in the HarfBuzz path. We can see that in the drawing function for shaped text, there is no call to a kerning function; instead, `advance_px = (HB x_advance) * scale` is computed directly [33] [35] . In contrast, in the fallback path for monospace or when HarfBuzz is off, the code calls `font_get_ttf_kerning(prev, curr)` and adds that to the pen position [36] . This conditional approach prevents double-applying kerning. Therefore, one *cause* of spacing issues – double kerning or missed kerning – is handled properly.

However, there is a nuance: SDL_ttf (FreeType) and HarfBuzz may sometimes disagree on glyph advance widths, especially with hinted fonts. WORR-2 addressed this by favoring SDL_ttf's metrics for each glyph's width when available, even in the HarfBuzz path. The docs mention that the glyph cache *"now prefers SDL_ttf bitmap metrics when a glyph is nominally mapped to a single codepoint"*, aligning advances and bearings with the actual rendered glyph [14] . In practice, after shaping, they call `font_get_ttf_glyph_index()` for each HarfBuzz glyph ID, which loads the glyph via SDL_ttf and caches its `glyph.advance` (in pixels) [37] [38] . They then likely adjust the HarfBuzz advance to match if needed. This prevents slight spacing errors where HarfBuzz might have a fractional advance and SDL_ttf (after hinting) has a rounded value. If this synchronization weren't done, spacing issues could manifest as some characters appearing a bit too tight or loose relative to others (especially noticeable in monospaced contexts or aligned text columns).

- **Letter Spacing (Tracking):** WORR-2 supports additional letter spacing via `font->letter_spacing`. This is a user-defined extra space (in pixel units of the font) to insert between characters. The code implements this by adding an offset **after** each glyph's advance in the render loop. Specifically, `if (!first_in_line && font->letter_spacing > 0) pen_x += (base_units * metric_scale * letter_spacing)` [39] [40] . In simpler terms, it adds a constant extra gap between glyphs. The same logic is applied in the text measurement function so that the computed width matches what will be drawn [41] [42] . Given this, letter spacing should not cause inconsistencies between measuring and drawing – they both include it. A potential issue is how letter spacing interacts with kerning and shaping. WORR-2's approach is to apply letter spacing on top of whatever spacing HarfBuzz or the font metrics provide. This means even if two letters are kerned closely by the font (say, "AV"), the engine will still add, for example, 1 pixel of spacing between them if letter_spacing is 1. That is usually fine (it just uniformly expands the text), but it could negate very tight kerning. If letter_spacing is being used (perhaps for certain UI themes), developers should be aware that it overrides the font's kerning to that extent. In terms of bugs, the implementation is straightforward and likely correct; any "spacing issue" related to this would be more about design preference than malfunction. For completeness, one might consider not adding letter spacing after a character that was combined into a ligature, but since letter spacing is added per glyph in the final shaped output, a ligature glyph still only gets one spacing after it, which is logically consistent.

- **Monospace and Fixed Advances:** WORR-2 tries to support monospace rendering (like the console font) even with TTF. For a fixed-width font, they bypass HarfBuzz layout and instead use each glyph's `font->fixed_advance` value [43]. This is set to the width of the character "M" (or a specified widest character) during font load [44] [45]. The idea is to ignore kerning and ligatures entirely and force each character to occupy the same advance. The code sets `font->ttf.fixed_advance_units` by measuring 'M' via SDL_ttf, then computes `font->fixed_advance = fixed_advance_units * (virtual_line_height / extent)` [46] [47]. At runtime, if `font->fixed_advance > 0`, the HarfBuzz path is skipped and each character is rendered in a loop with equal spacing [43]. This ensures console text alignment (columns of text line up, etc.). The known issue here was that enabling HarfBuzz could disturb monospace spacing (e.g., some characters might form unintended ligatures or spacing might drift). WORR-2 solved it by the explicit bypass. So, fixed-width fonts should not suffer spacing anomalies now. Any **remaining problem** in console spacing likely comes from the baseline or scaling, not horizontal spacing, since monospacing is enforced.

- **Fractional Advances and Pixel-Snapping:** One persistent source of "odd" spacing can be fractional positions. Although WORR-2 mostly operates in virtual coordinates (which are integers) scaled by integer factors, the introduction of the 1.5× scale and HarfBuzz's potential fractional advances (in FreeType, advances are in 26.6 fixed-point units) means the pen position `pen_x` can become non-integer in virtual units. For example, if a font advance is 5.5 virtual pixels, the engine will accumulate that and later floor it when drawing to actual pixels. This can lead to some glyphs rounding differently than others, causing uneven gaps (one gap floor to 5 px, another to 6 px). WORR-2 anticipated this and added a toggle `cl_font_ttf_hb_snap` to snap positions to nearest integer [48]. When enabled, it rounds each glyph's position (`pen_x`, `bearing_x`, `x_offset`, etc.) to int before computing the next glyph's draw position [40] [34]. Snapping eliminates fractional discrepancies at the cost of slightly altering the designed spacing. By default this is off (`cl_font_ttf_hb_snap = 0`) presumably to preserve exact kerning ratios. If the spacing issues are visible (e.g., some character pairs look a bit too tight or too loose by a single pixel), enabling snapping might actually improve the overall consistency, given the virtual screen's pixel-perfect goals. In other words, a user complaining about spacing might find that with snapping on, the text looks more uniformly spaced (no subpixel blurriness or one-pixel deviations). As a developer, one should decide if the tiny typographic benefit of subpixel positioning is worth the potential visual inconsistency. In a retro-style or pixel-aligned UI, snapping is often preferred.

- **Overall Text Block Alignment:** Spacing issues aren't only between letters; they also concern aligning whole strings. For instance, centering text or right-aligning text can reveal if the measured width is accurate. WORR-2's `Font_MeasureString` returns an integer width (rounded from the floating calculation) [49] [50]. If fractional advances accumulate to, say, X.9, it will round up to X+1 in width. But when drawing the same text, if those fractions caused rounding down at each glyph, the drawn width might end up slightly less. There is a debug dump (`cl_font_debug_dump 1`) that prints detailed metrics of a rendered string, including the measured width and actual drawn positions [51] [52]. Using this, one can detect if there's a discrepancy. Typically, WORR-2 tries to Q_rint (round) the final `last_x` pen position to get the text width [53], which should match the drawn pixels. If any inconsistency remains, it could cause slight misalignment when, say, centering text (the text might render a half-pixel off center). Given the available code, this is probably handled, but it's something to verify when addressing spacing complaints.

In summary, WORR-2's horizontal spacing logic is quite robust: it uses HarfBuzz for correctness and supplements it with its own metrics and optional pixel-snapping for stability. The main areas to watch are whether to enable snapping (for pixel alignment) and ensuring that letter spacing or other adjustments don't conflict with the intended kerning. "Completely wrong" spacing might be an exaggeration, but any observed spacing oddities can likely be fixed by one of the above mechanisms (e.g., toggling snap or adjusting the letter spacing value).

Implementation Plan for Fixes and Enhancements

To ensure the font implementation is as good as it can be, especially given the virtual screen setup, we propose a detailed plan addressing the above issues:

1. Fine-Tune Baseline Alignment: We should calibrate the baseline so that text aligns perfectly with UI elements like the console cursor and menu graphics. Using the built-in debug overlay for baselines [31] [32], we will adjust the baseline offset if necessary. This could involve tweaking the `rendered_ascent` computation. For example, if we observe that the green baseline line sits a pixel too high relative to the console underscore (meaning characters seem low), we could reduce `font->ttf.baseline` by 1 when computing metrics. This effectively raises all glyphs by 1 pixel. Conversely, if text seems too high (baseline line cutting through lower half of characters), we might increase the baseline metric. These adjustments can be done in `font_compute_ttf_metrics` or right after `font_compute_ttf_rendered_metrics`. Because WORR-2 already ties baseline to line height ratio, a one-pixel change in baseline will consistently propagate to all lines. We will document this as a font-specific tweak (some fonts may need a nudge depending on their design). The goal is to lock the baseline so that, in the debug overlay, the baseline guide (green) exactly coincides with where we conceptually want the baseline (e.g., the bottom of non-descending letters). After this change, the legacy console prompt (`conchars.png` glyph) and the TTF text will sit on the same line [7] [26].

2. Broaden Glyph Preloading for Metrics: To avoid any runtime surprises with baseline shifts, we will preload a more extensive set of glyphs at font load. Currently, ASCII 32–126 are preloaded [54]. If the game will use other Unicode ranges (Latin-1 accented letters, Cyrillic, etc.), we should preload at least the tallest glyphs from those ranges. For Latin scripts, characters like Á (A-acute) or Þ (Thorn) might have larger extents. We can extend `font_preload_ascii()` to `font_preload_basic_latin()` and include 32–255 or specific ranges. Then call `font_compute_ttf_rendered_metrics()` on that superset. This way, `rendered_ascent` and `rendered_descent` truly reflect the maximum extents of any likely character. WORR-2 already handles on-the-fly metric growth [27], but preloading ensures consistent baseline from the start, which is cleaner. We must be mindful of memory (preloading hundreds of glyphs), but since most UI fonts are small and ASCII, this is usually fine. We will also keep the dynamic update as a fallback (for modders who might use an unusual glyph).

3. Validate Line Height Consistency: We will verify that the chosen line height (Font_LineHeight which is roughly 1.5× virtual height) is optimal. If the UI design expects a different spacing (for example, exactly 8 pixels with no extra gap, or conversely more gap), we have two options: adjust `k_font_scale_boost` or adjust `pixel_spacing`. The current design uses an integer base scale and a `pixel_spacing = 1/pixel_scale` (usually 1) to guarantee a minimal gap [55]. This ensures, for instance, that underlining from one line doesn't touch descenders of the line above. We will confirm that `pixel_spacing` of 1 is enough by checking in the debug overlay that the red line of one line and the blue line of the next line do not overlap [32]. If the font has very tall accents, possibly a gap of 2 might be needed at certain scales – we can expose a cvar for line spacing if needed. However, since the current spacing seems to intentionally mimic the original console (which likely had 1 pixel between lines), we will likely keep it. In short, no overlapping and visually balanced line gaps will be our criteria.

4. Consider Removing the 1.5× Scale Boost: The `k_font_scale_boost = 1.5` is a somewhat arbitrary factor. It was probably introduced so that an 8px font renders at 12px to better match old-school readability or aspect ratio. We will evaluate whether this is still necessary. One approach is to remove the 1.5 multiplier and instead load the font at a larger pixel_height. For example, for the console font, we could load at 12 pixels and use scale 1.0. If we do this, all metric calculations would simplify (no mysterious 1.5 factor in every formula). The risk is that this could alter alignment if something was tuned expecting 1.5. We'll perform an experiment with the console font: load at pixel_height = 12 * supersample (instead of 8 * supersample) and set k_font_scale_boost = 1.0, then compare the alignment. If everything lines up equally or better, we can adopt that change. It would make reasoning about metrics easier (virtual_line_height would directly correspond to actual line height). If differences arise, and given that the code is working now, we might keep the 1.5 and just clearly comment its purpose. Should we keep it, we ensure all relevant places (line height, baseline calc, etc.) include it – which they currently do [56] [30]. In either case, this step is about simplifying the system to reduce "unknown" offsets that could cause alignment issues.

5. Enable and Test Pixel Snapping: Since the virtual screen uses integer scaling, pixel perfection is important. We will test turning on `cl_font_ttf_hb_snap` (set it to 1) and examine text in various UI screens. Pay special attention to any letters with obvious blur or off-by-one spacing in unsnapped mode, and see if snapping fixes them. For example, italicized text or certain kerning pairs might show a 1px shift when moving. Snapping should eliminate that by ensuring each glyph starts on an integer position. We must ensure that this doesn't reintroduce any baseline jitter; the snap code also snaps vertical offsets (`y_offset`, which for normal Latin text is usually 0) [34]. Since most of our fonts are small and hinted, snapping likely improves clarity. If our tests confirm overall better appearance, we could default this to on (or even remove the option and always snap). We'll document this choice: essentially trading subpixel positioning for consistent pixel alignment, which fits the project's aesthetic. Any *spacing* "wrongness" perceived by the user could very well be solved by this, as a glyph that was say 7.5 pixels apart from the next would sometimes render 7px apart and sometimes 8px due to rounding – snapping makes it always 8px, for instance. That consistency might be what the user expects.

6. Review Color Code and Cluster Handling: WORR-2 strings often include color escape codes (like `^1` for red). The shaping code has to strip these for HarfBuzz, then reapply colors per cluster [57] [58]. We should verify that this system isn't inadvertently affecting spacing. The code uses `hb_buffer_cluster` values to map glyphs back to the original string indices [59] [60]. If a color code splits a cluster (say, in the middle of a multi-byte UTF8 sequence or a ligature), the text could be split incorrectly. The current implementation tracks color changes outside of the shaping (it reconstructs the nominal codepoint sequence without color markers). It appears robust, but as a sanity check, we should test a string like "^1fi^7" (where "fi" might form a ligature). Make sure the presence of a color code does not prevent the ligature or affect spacing. The expected behavior is that the ligature might not form across a color boundary – which is fine – but each part should still be correctly positioned. If any anomaly is found, we might need to refine how we handle cluster breaks at color boundaries (perhaps treat a color code as a text break for shaping purposes). This is a minor detail, but worth ensuring since mis-spaced text could be blamed on the font when it's actually color-code logic.

7. Comprehensive Testing Across UI: Finally, once tweaks are made, we'll do a pass through all UI elements: console, chat box, menus, HUD indicators, etc. We will use the debug overlay lines (guides for each line's top, baseline, bottom) to confirm alignment. For the console and chat, ensure the baseline aligns with any graphics (like the input prompt '>' or underscore). For menu buttons, ensure text is centered within its box as expected (the baseline of a capital letter might visually need centering adjustments, since fonts have inherent ascent/descent differences – sometimes centering text requires a slight vertical offset). We might find we need to offset text in certain UI elements by a couple pixels to look optically centered, even if mathematically centered. Such adjustments can be done in the UI layout definitions. The key is that the font rendering itself provides stable, predictable metrics so we can apply those offsets consistently. We will also test multiple resolutions and aspect ratios to verify that the integer scaling doesn't introduce new issues – it shouldn't, since all coordinates scale uniformly. If the text was aligned at 640×480 virtual, it will remain aligned when scaled 2× or 3×, etc., as long as our baseline and spacing are based on those same virtual units (which they are). Any fractional discrepancies at one scale won't accumulate because of the snapping and rounding.

Throughout this process, we'll document the changes and results in the project's `doc/` folder (similar to previous entries like *ttf-baseline-alignment* and *ttf-debug-tools*). This will help future developers understand the reasoning behind the adjustments. By following these steps, we expect to eliminate the "wrong" font alignment: text will sit on the baseline as intended, with equal spacing and no overlap or odd gaps, across all elements of WORR-2's interface. The TrueType fonts will effectively mimic the alignment consistency of monospaced bitmap fonts, while retaining proper proportional spacing and high-quality rendering.

---

[1] [2] [3] FreeType Glyph Conventions | Glyph Metrics
https://freetype.org/freetype2/docs/glyphs/glyphs-3.html

[4] c++ - Rendering TrueType fonts as fixed size (like in a terminal emulator) - Stack Overflow
https://stackoverflow.com/questions/76840825/rendering-truetype-fonts-as-fixed-size-like-in-a-terminal-emulator

[5] [6] [10] [11] [12] [21] [29] [30] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [44] [45] [46] [47] [49] [50] [51] [52] [53] [54] [55] [56] [58] [59] [60] font.cpp
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/src/client/font.cpp

[7] [8] [22] [23] [26] ttf-baseline-alignment-2026-01-20.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-baseline-alignment-2026-01-20.md

[9] [31] ttf-debug-tools-2026-01-27.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-debug-tools-2026-01-27.md

[13] [28] [57] ttf-harfbuzz-shaping-2026-01-22.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-harfbuzz-shaping-2026-01-22.md

[14] [27] [48] ttf-hb-metrics-debug-2026-01-27.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-hb-metrics-debug-2026-01-27.md

[15] [43] ttf-layout-baseline-fix-2026-01-22.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-layout-baseline-fix-2026-01-22.md

[16] [17] [18] [19] [20] virtual-screen-2d-scaling.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/virtual-screen-2d-scaling.md

24 25 ttf-baseline-metrics-2026-01-20.md
https://github.com/themuffinator/WORR-2/blob/8192492a27f26ec7b819e19540b3e7835caff67b/doc/ttf-baseline-metrics-2026-01-20.md