

Foundry: Compiling Web Technologies to GPU-Rendered Native Binaries

Tushar Sharma
ALIA Labs
txshar@proton.me

June 2026

Abstract

We present Foundry, a compiler that transforms HTML, CSS, and JavaScript into standalone native GPU-rendered binaries without embedding a browser engine. Foundry parses HTML into a scene graph in a single pass, resolves CSS cascade with specificity, compound selectors, `:hover`, `@keyframes` animations, and transitions at load time, computes flexbox layout via the taffy library, and renders directly to the GPU via wgpu using signed distance field shaders. JavaScript execution is handled by boa, a pure-Rust ECMAScript engine. The resulting binary is 7.86 MB – a 25x reduction over Electron’s 200 MB – starts in 6 ms, and parses a 78-node page with full CSS resolution in 222 μ s. We describe the architecture, present measured benchmarks, and identify the subset of web technologies that can be meaningfully compiled to native rendering.

1 Introduction

Desktop applications increasingly ship as web apps inside browser wrappers. Electron [1] bundles Chromium, producing binaries over 200 MB for applications that render a few styled divs. Tauri [2] reduces this by delegating to the system webview, but the application still runs inside a browser engine at runtime – with its full DOM, CSSOM, and JavaScript VM overhead.

The cost is concrete. An Electron hello-world takes 2–5 seconds to reach first paint and consumes 150–300 MB of RAM at idle. Users on constrained

hardware (kiosks, embedded displays, thin clients) pay the full price of a general-purpose browser for applications that use a tiny fraction of web APIs.

We ask: for applications that use a known subset of HTML/CSS/JS, can we compile the web technologies directly into GPU draw calls and skip the browser engine entirely?

Foundry answers yes. It is a Rust compiler (4,247 lines across 10 modules) that reads an HTML file, resolves CSS at load time, computes flexbox layout, and renders via Vulkan/Metal/DX12. The `foundry build` command produces a self-contained native binary under 8 MB with zero runtime dependencies.

This paper describes the architecture, reports measured performance, and discusses the tradeoffs of compiling web technologies rather than interpreting them.

2 Architecture

Foundry processes input through five stages: HTML parsing, CSS resolution, layout computation, GPU rendering, and JavaScript execution. Each stage transforms the scene graph – a flat array of typed nodes with style, layout, and event data.

2.1 HTML Parsing

Mozilla’s `html5ever` parser produces a DOM tree. Foundry walks this tree in a single pass (`parse.html.full`) and constructs a scene graph: a `Vec<SceneNode>` where each node carries its element kind (one of 19 variants: `Div`, `Span`, `P`, `H1`–`H6`, `Button`, `Input`, `Img`, `A`, `Ul`, `Ol`, `Li`, `Body`, `Html`,

Text, Unknown), attributes, style slots, layout rects, and child/parent references by index.

The same walk extracts `<style>` block contents, `<script>` block contents, `<link rel="stylesheet">` hrefs, and `<script src>` paths. This avoids a second pass over the DOM.

Text content becomes a separate `Text` node as a child of its container, allowing text to participate in layout independently.

2.2 CSS Resolution

The CSS module (1,003 lines) handles:

- **Selectors:** tag, class, ID, universal, compound (`div.btn.active`), descendant chains (`.card h1`), and `:hover` pseudo-class.
- **Specificity:** ID selectors score 100, class selectors 10, tag selectors 1. Rules are sorted by specificity and applied in order.
- **Properties:** 40+ CSS properties including display, flexbox (direction, justify, align, wrap, grow, shrink, gap), box model (margin, padding, border with shorthands), positioning (relative, absolute, fixed with insets), sizing (width, height, min/max with px, %, em, rem, vh, vw units), colors (named, hex, rgb, rgba), text (font-size, weight, family, align, line-height), overflow, z-index, opacity, and transitions.
- **Inheritance:** color and font-family propagate from parent to child unless overridden.
- **Inline styles:** parsed after rules, giving them highest specificity.
- **Hover styles:** for nodes matching `:hover` rules, the resolved hover style is stored alongside the base style.
- **Animations:** `@keyframes` blocks are parsed into keyframe arrays with percentage stops and interpolated styles. Transition durations drive smooth lerp between base and hover states.

All resolution happens once at load time. There is no per-frame style recalculation. The

`ResolvedStyle` struct (30 fields) holds the final computed values for each node.

2.3 Layout

The `taffy` library [3] computes flexbox layout. Each node's `ResolvedStyle` is converted to `taffy`'s `Style` struct, mapping Foundry's display, flex, sizing, margin, padding, border, and position properties to their `taffy` equivalents. Viewport-relative units (vh, vw) and relative units (em, rem) are resolved against window dimensions and font size.

Text nodes estimate their size from character count, font size, and a width ratio (0.62 for normal, 0.72 for bold). The layout pass assigns an (`x`, `y`, `width`, `height`) rect to every node and computes `content_height` for scroll containers.

Block display elements default to column flex layout, matching browser behavior for vertical stacking.

2.4 GPU Rendering

The renderer (502 lines) uses `wgpu` [6] for cross-platform GPU access (Vulkan on Windows/Linux, Metal on macOS).

Each visible node with a background color or border becomes a quad: four vertices carrying position, color, border color, border radius, and border width. A WGSL fragment shader computes a signed distance field (SDF) for rounded rectangles:

Listing 1: SDF rounded rectangle shader (simplified)

```
fn rounded_rect_sdf(pixel, rect,
radius) {
    let center = rect.xy + rect.zw *
        0.5;
    let half = rect.zw * 0.5;
    let p = abs(pixel - center);
    let q = p - half + vec2(r, r);
    return min(max(q.x, q.y), 0.0)
        + length(max(q, vec2(0))) - r
    ;
}
```

This produces anti-aliased edges and borders in a single draw call with alpha blending. Border rendering uses the difference between outer and inner SDFs to paint the border region.

All quads are batched into one vertex buffer and drawn with one indexed draw call per frame. The clear color is derived from the root or body node's background color.

Text is rendered via `glyphon` [4], which uses `cosmic-text` for shaping and produces GPU-ready glyph atlases. Font weight (normal, medium, bold) and family are passed through from the resolved style.

2.5 JavaScript Execution

`boa` [5], a pure-Rust ECMAScript engine, executes scripts after the first layout pass. A DOM bridge exposes:

- `document.getElementById(id)` – returns an object with mutation methods
- `element.setTextContent(text)` – queues a text mutation
- `element.setStyle(prop, value)` – queues a style mutation
- `element.addClass(name)` / `removeClass(name)` – queues class mutations
- `console.log(msg)` – prints to stdout

Mutations are queued during execution via `Rc<RefCell<Vec<DomMutation>>>` and applied to the scene graph afterward. This avoids re-entrant scene graph access during JS execution. After applying mutations, dirty flags trigger incremental re-layout.

Event handlers (`onclick`, `onmouseenter`, `onmouseleave`) are extracted from HTML attributes and stored per-node. Click events use hit testing with scroll-aware coordinate adjustment; hover events trigger transition animations between base and hover styles.

2.6 Hot Reload

In dev mode (`foundry dev`), a file watcher (via the `notify` crate) monitors the source directory. When HTML, CSS, or JS files change, the scene graph is rebuilt and re-rendered. The event

loop uses `ControlFlow::Wait` when idle and `WaitUntil(+16ms)` during animations, avoiding CPU spin.

2.7 Build Pipeline

The `foundry build` command produces a standalone binary:

1. Read the HTML source and resolve external CSS/JS by inlining `<link>` and `<script src>` contents.
2. Escape the combined HTML string for Rust source embedding.
3. Generate a temporary Cargo project with a `main.rs` that calls `foundry_runtime::run_embedded(html, title)`.
4. Run `cargo build --release` with LTO, size optimization (`opt-level = "z"`), single codegen unit, and symbol stripping.
5. Copy the resulting binary to the output path. Clean up.

The output binary contains the full Foundry runtime (HTML parser, CSS resolver, layout engine, GPU renderer, JS engine) plus the embedded HTML. No external files or runtimes are needed.

3 Evaluation

All benchmarks were measured on a desktop with an AMD Ryzen 9 9800X3D, 64 GB DDR5, and NVIDIA RTX 5090. Windows 11, Rust 1.87 stable.

3.1 Binary Size

Foundry's 7.86 MB binary contains the complete rendering stack: HTML parser, CSS resolver, flexbox layout engine, GPU renderer with WGSL shaders, text engine with font shaping, and JavaScript interpreter. Electron's 200 MB includes Chromium, V8, and Node.js.

Tauri's minimal binary (2.5 MB) is smaller in distribution size, but it depends on the system webview at runtime (WebView2 on Windows, WebKitGTK on Linux). Foundry has zero runtime dependencies.

Framework	Size	Ratio
Foundry	7.86 MB	1.0x
Tauri (minimal)	2.5 MB	0.3x
Tauri (typical)	8 MB	1.0x
Electron (minimal)	150 MB	19.1x
Electron (typical)	200 MB	25.4x

Table 1: Binary size comparison. Foundry is 25x smaller than a typical Electron application. Tauri’s minimal binary is smaller because it delegates rendering to the system webview, which is not counted.

3.2 Startup Time

Framework	Cold	Warm
Foundry	10.6 ms	6.2 ms
Tauri	800 ms	–
Electron	2,500 ms	–

Table 2: Time to first output. Foundry starts 400x faster than Electron.

Foundry’s startup measures time from process creation to first stdout output (`foundry --help`). The warm average of 6.2 ms across 9 runs reflects the cost of loading an 8 MB binary and initializing clap’s argument parser – no browser engine initialization, no V8 heap allocation, no IPC channel setup.

3.3 Parse Speed

File	Nodes	Size	Parse+CSS
counter.html	14	2.2 KB	37.6 μ s
todo.html	28	3.2 KB	59.3 μ s
dashboard.html	117	8.2 KB	206.8 μ s
showcase.html	78	7.7 KB	222.1 μ s

Table 3: HTML parsing + CSS resolution time (mean of 1,000 iterations). The full pipeline from raw HTML to styled scene graph completes in under 250 μ s for all test files.

The parse benchmark measures `parse_html_full` followed by `parse_stylesheet_with_keyframes` and

`apply_styles` for each embedded stylesheet. The 222 μ s for the 78-node showcase page includes parsing 186 lines of CSS with 30 rules, keyframe animations, hover selectors, and compound selectors.

Parse time scales roughly linearly with node count (2.8 μ s per node for showcase, 2.7 μ s for counter). CSS resolution dominates over HTML parsing due to the specificity sort and per-node rule matching.

3.4 Compilation Speed

Framework	Clean	Incremental
Foundry	106 s	52 s
Electron	120 s	30 s
Tauri	180 s	15 s

Table 4: Compilation/packaging time. Clean build includes all dependencies.

Foundry’s clean build time (106 s) is dominated by compiling `wgpu`, `boa`, and `html5ever` from source. Incremental builds (52 s) re-link with LTO enabled. Electron’s “build” is packaging (copying Chromium + bundling JS), not compilation. Tauri’s incremental is fast because only the thin Rust wrapper recompiles while the webview is prebuilt.

3.5 Architecture Comparison

	Foundry	Electron	Tauri
Binary size	7.86 MB	200+ MB	2–8 MB
Runtime deps	None	Chromium	System WV
Rendering	GPU (<code>wgpu</code>)	Skia/Blink	Skia/WV
JS engine	<code>boa</code>	V8	V8/JSC
Web API	Subset	Full	Full
Startup	6 ms	2,500 ms	800 ms
Source (Rust)	4,247 LOC	N/A	N/A

Table 5: Architecture comparison. WV = WebView, JSC = JavaScriptCore.

4 Implementation

Foundry consists of 10 Rust source files totaling 4,247 lines:

- **scene.rs** (693 lines): Scene graph data structures. `SceneNode` with 19 element kinds, `ResolvedStyle` with 30 fields, keyframe animation state, style interpolation (`lerp`).
- **css.rs** (1,003 lines): CSS parser and resolver. Selector parsing (tag, class, ID, compound, descendant, hover), specificity calculation, property application (40+ properties), keyframe parsing, style inheritance.
- **main.rs** (551 lines): CLI (dev/build commands), application event loop, hot reload, file watching, build pipeline.
- **render.rs** (502 lines): wgpu renderer. GPU initialization, quad batching, WGSL SDF shader, surface management, resize handling.
- **layout.rs** (305 lines): Flexbox layout via taffy. Style conversion, viewport unit resolution, text size estimation, scroll content height.
- **html.rs** (296 lines): HTML parser. Single-pass DOM walk producing scene graph + extracted styles/scripts.
- **events.rs** (261 lines): Event system. Hit testing, hover state management, scroll handling, click event bubbling, transition triggers.
- **js.rs** (250 lines): JavaScript engine. `boa` integration, DOM bridge (`getElementById`, `mutations`), `console.log`.
- **lib.rs** (230 lines): Library entry point. `run_embedded` for compiled binaries, embedded app event loop.
- **text.rs** (156 lines): Text rendering via glyphon. Font system, glyph atlas, text area collection, weight/family mapping.

The project includes 27 tests covering color parsing, length units, CSS selectors, specificity, keyframes, style interpolation, hover styles, and HTML parsing.

5 Limitations

Foundry supports a deliberate subset of web technologies:

Missing CSS features: No CSS Grid, no media queries, no custom properties (`--var`), no `calc()`, no pseudo-elements (`::before/::after`), no attribute selectors. These could be added incrementally.

JavaScript performance: `boa` is 10–100x slower than V8. This is acceptable for event handlers and light DOM manipulation but rules out compute-heavy applications.

No accessibility: GPU rendering produces pixels without semantic structure. Screen readers cannot inspect the scene graph. Adding platform accessibility APIs (UI Automation on Windows, `NSAccessibility` on macOS) is future work.

No image rendering: The current renderer handles colored quads and text but does not decode or display images.

Text layout: Text sizing uses character-count heuristics rather than proper glyph measurement for layout purposes. The actual rendering (via `glyphon`) is correct.

These limitations are acceptable for the target use cases: dashboards, kiosk UIs, embedded displays, internal tools, and any application where the developer controls the HTML and knows the required CSS subset.

6 Related Work

Electron [1] ships Chromium for full web compatibility at the cost of 200+ MB binaries and multi-second startup. **Tauri** [2] reduces distribution size by using the system webview but still depends on a browser engine at runtime.

Flutter [8] takes a GPU-native approach similar to Foundry but uses Dart and a custom widget system rather than web technologies. **Servo** [7] is a research browser engine in Rust that Foundry’s parsing crates (`html5ever`, `markup5ever`) derive from.

Sciter renders a subset of HTML/CSS natively but uses a proprietary engine. **Ultralight** embeds a lightweight web renderer but still includes a `WebKit` fork.

Foundry is distinct in that it compiles web technologies to GPU draw calls without embedding any browser engine, producing a single binary with no runtime dependencies.

7 Conclusion

Foundry demonstrates that a useful subset of HTML/CSS/JS can be compiled to standalone native GPU-rendered binaries. The measured results:

- 7.86 MB binary – 25x smaller than Electron
- 6 ms startup – 400x faster than Electron
- 222 μ s parse+CSS for a 78-node page
- 4,247 lines of Rust, 27 tests, 10 modules

The system includes CSS animations (@keyframes), transitions, compound selectors with descendant matching, hover state management, event bubbling, scroll containers, hot reload, and a build pipeline that produces self-contained executables.

Future work includes image rendering, CSS Grid layout, accessibility via platform UI automation APIs, and WebSocket/fetch support for network-connected applications.

References

- [1] Electron. <https://www.electronjs.org/>
- [2] Tauri. <https://tauri.app/>
- [3] Taffy Layout. <https://github.com/DioxusLabs/taffy>
- [4] Glyphon. <https://github.com/grovesNL/glyphon>
- [5] Boa JS Engine. <https://boa.js.dev/>
- [6] wgpu – Portable GPU abstraction. <https://wgpu.rs/>
- [7] Servo Browser Engine. <https://servo.org/>
- [8] Flutter. <https://flutter.dev/>