Bridging the Gap Between Browser and Backend Media Processing

Romain Beauxis

Descript

FOSDEM Open Media Devroom 2026

Outline

Introduction

- Web applications are increasingly media-rich
- Browser APIs: Canvas, WebGL, WebCodecs, WebAudio, WebRTC, WebAssembly
- Challenge: How to create a top-of-the art in-browser experience??
- Real-world examples from building Descript, a web-based video editor

About Me

Background:

- Involved in media and streaming since college (École Centrale, VLC)
- Key contributions:
  - **Liquidsoap:** Media streaming language for radio stations and live streaming
  - **FFmpeg:** Tentative contributor to the open-source multimedia framework
- Professional web development for over a decade
- Audio media experience at AudioSocket

Current Role:

- Merging web development expertise with media streaming knowledge at Descript
- Building browser-based video editing tools with backend processing

Descript: The Evolution

2017: Audio-First

- Text-based audio editing interface
- Workflow: Transcribe audio → edit transcript → audio modified accordingly
- Delete words from transcript removes corresponding audio segments
- Targets podcast and audio content production

2019-Present: Video + AI

- Extended text-based editing to video content
- AI-based features added:
  - Overdub (neural voice synthesis)
  - Studio Sound (audio processing pipeline)
  - Eye Contact (face detection + warping)
  - Background removal (segmentation)
- Web-based editor launched 2023

Key architectural principle: Media elements addressable via transcript, enabling text-based manipulation of audio/video timeline

Current Product: Full-Stack Media Editor

Core Capabilities

- **Transcription:** Real-time speech-to-text with speaker diarization
- **Text-Based Editing:** Cut, copy, paste video by editing transcript
- **Multi-Track Timeline:** Traditional timeline view + layers
- **AI Effects:** Voice cloning, audio enhancement, visual effects
- **Collaboration:** Real-time co-editing (like Google Docs for video)
- **Publishing:** Export to multiple formats, platforms

Platform:

- Desktop app (Electron)
- Web app (launched 2023)
- Cloud backend for heavy processing

AI Features: Processing Architecture

Client-Side Processing

- **Filler Word Removal**
  Regex pattern matching on transcript
- **Basic Cuts**
  Timeline data structure updates
- **Preview Playback**
  WebCodecs decode + Canvas render

Backend Processing

- **Overdub (Voice Synthesis)**
  Neural TTS inference
- **Studio Sound**
  Multi-stage FFmpeg audio pipeline
- **Eye Contact**
  Face landmark detection + image warping
- **Background Removal**
  Semantic segmentation models

Integration challenge: Managing state transitions between client and backend processing contexts

Understanding the capabilities and limitations of browser-based media processing

Browser Media APIs Overview

Modern browsers provide specialized APIs for media processing:

Core APIs

- **Web Audio API:** Real-time audio processing graph
- **WebCodecs:** Low-level encode/decode (since 2022)
- **WebGL/Canvas2D:** Video effects and text rendering

Challenge: Format support varies significantly by browser vendor

- Apple platforms: HEVC, ProRes (hardware acceleration)
- Chromium/Firefox: VP8, VP9, AV1 (open codecs)
- H.264: Universal baseline compatibility

How to maintain consistent UX across different browsers, platforms, and collaborative sessions?

Format Support: Codec Variability

Format	Chrome/Edge	Safari	Firefox	FFmpeg	WebCodecs
H.264 (AVC)	✓	✓	✓	✓	✓
H.265 (HEVC)	∼	✓	∼	✓	✓
HEIC	×	✓	×	✓	×
VP8	✓	∼ (WebRTC)	✓	✓	✓
VP9	✓	✓	✓	✓	✓
AV1	✓	∼ (HW req)	✓	✓	✓
ProRes	×	✓	×	✓	×
AAC	✓	✓	✓	✓	×
Opus	✓	✓	✓	✓	×
MP3	✓	✓	✓	✓	×

Key observations:

- **FFmpeg:** Universal support for all codecs
- **WebCodecs:** Video codecs only (H.264, HEVC, VP8, VP9, AV1)
- **AV1 Safari:** Requires hardware decoder (iPhone 15 Pro, M3 MacBook Pro+)
- **Third-party browsers:** Chrome on iOS uses WebKit (Safari engine)

Format Support: Container Formats

Container	Chrome/Edge	Safari	Firefox	FFmpeg
MP4	✓	✓	✓	✓
WebM	✓	✓	✓	✓
Matroska (MKV)	✓	×	∼	✓
MOV	∼	✓	∼	✓
MPEG-TS	∼ (HLS)	∼ (HLS)	×	✓
Ogg	✓	∼	✓	✓

Practical implications:

- **MP4:** Safest choice for cross-browser compatibility
- **FFmpeg:** Universal container support on backend
- **Container support depends on underlying codecs:** A supported container with unsupported codec will fail
- **Unsupported containers:**
  - If codecs are browser-compatible: Remux only (client-side with libav.js, etc.)
  - If codecs incompatible: Full transcode required (typically backend)
- **General strategy:** Check codec compatibility first, then decide remux vs transcode

Audio API: WebCodecs AudioDecoder

Low-level audio decoding with EncodedAudioChunk:

```
const decoder = new AudioDecoder({
  output: (audioData) => {
    // Access: format, sampleRate, numberOfChannels, numberOfFrames, timestamp
    const buffer = new Float32Array(audioData.numberOfFrames *
                                    audioData.numberOfChannels);
    audioData.copyTo(buffer, { planeIndex: 0, format: 'f32-planar' });
    audioData.close();
  },
  error: (e) => console.error(e)
});

decoder.configure({ codec: 'opus', sampleRate: 48000, numberOfChannels: 2 });

// Create and decode encoded audio chunk
const chunk = new EncodedAudioChunk({
  type: 'key',              // 'key' or 'delta'
  timestamp: 0,             // Microseconds
  data: encodedDataBuffer   // ArrayBuffer from demuxer
});
decoder.decode(chunk);
```

Key advantage: Direct access to PCM data and precise timing information

Video API: WebCodecs VideoDecoder

Low-level video decoding with EncodedVideoChunk:

```
const decoder = new VideoDecoder({
  output: (frame) => {
    // Access: codedWidth/Height, displayWidth/Height, visibleRect,
    //         format, colorSpace, timestamp, duration
    ctx.drawImage(frame, 0, 0);
    frame.close();
  },
  error: (e) => console.error(e)
});

decoder.configure({ codec: 'avc1.42E01E', codedWidth: 1920, codedHeight: 1080 });

// Create and decode encoded video chunk
const chunk = new EncodedVideoChunk({
  type: 'key',              // 'key' or 'delta'
  timestamp: 0,             // Microseconds
  data: encodedDataBuffer   // ArrayBuffer from demuxer
});
decoder.decode(chunk);
```

Key advantage: Pixel format, color space, timing, visible rect for frame manipulation

Container Operations: No Native Muxing/Demuxing API

Browsers lack native APIs for container manipulation

WebCodecs API (2022): Encode/decode only, *no muxing/demuxing*

From MDN: "There is currently no API for demuxing media containers"

Solution: JS/WASM libraries (libav.js, etc.)

- Complex integration (binary formats, offsets, flags)
- Performance overhead vs native
- Limited format support per library

Impact: Must handle muxing/demuxing client-side or offload to backend

Rendering: WebGL for GPU Effects

Real-time video effects with shaders:

```
const gl = canvas.getContext('webgl2');
const program = createShaderProgram(gl, vertexShader, fragmentShader);

// Upload video frame as texture
const texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
              videoFrame);  // VideoFrame, Canvas, or ImageBitmap

// Render with custom shader
gl.useProgram(program);
gl.drawArrays(gl.TRIANGLES, 0, 6);

// Read pixels back
const pixels = new Uint8Array(width * height * 4);
gl.readPixels(0, 0, width, height, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

Performance: GPU acceleration enables 60fps effects at 1080p/4K

Rendering: Canvas2D for Text

Sophisticated text rendering with web fonts:

```javascript
const canvas = document.createElement('canvas');
canvas.width = 1920;
canvas.height = 1080;
const ctx = canvas.getContext('2d');

await document.fonts.load('48px "Roboto"');  // Wait for font

// Advanced typography support
ctx.font = '48px "Roboto", sans-serif';
ctx.fillStyle = 'white';
ctx.strokeStyle = 'black';
ctx.textAlign = 'center';

ctx.strokeText('Hello World', 960, 540);
ctx.fillText('Hello World', 960, 540);

// Supports: Web fonts, emoji, ligatures, kerning, font features
```

Advantage: Browser text rendering far more sophisticated than FFmpeg drawtext

Text Rendering: Cross-Platform Font Challenges

System font availability varies significantly across platforms

Emoji Rendering Discrepancies

Same Unicode codepoint (U+1F600 "grinning face") renders differently:

macOS/iOS
Apple Color Emoji

Android
Noto Color Emoji

Windows
Segoe UI Emoji

Samsung
One UI

Impact: Same emoji code can convey different emotions across platforms

Browser text rendering: Leverages platform fonts, emoji, ligatures, kerning
→ More sophisticated than FFmpeg drawtext, but less cross-platform predictable

Frame-Accurate Seeking in VFR Media

Challenges and solutions for precise timeline navigation in variable frame rate videos

The VFR Seeking Problem

Challenge: Fast, accurate seeking in Variable Frame Rate media

Variable Frame Rate (VFR) Characteristics

- Screen recordings, smartphone videos
- Irregular frame spacing: Frame 0: 0ms, 1: 33ms, 2: 66ms, 3: 99ms, 4: 150ms (dropped!)
- Timeline editors need frame index → timestamp mapping

Compressed Video Seeking Constraint

- **Keyframes (I-frames):** Self-contained, can decode independently
- **P/B-frames:** Depend on previous/future frames
- Typical GOP: I-frame every 1-2 seconds (30-120 frames apart)
- **Must seek to nearest prior keyframe, then decode forward**

Goal: Know exactly where keyframes are → seek instantly → decode to target frame

Hybrid Architecture (1/2): libav.js for Demuxing + Seeking

WASM FFmpeg extracts timing and positions at keyframes

Step 1: Demux + Extract Timing	**Step 2: Seek to Keyframe**
MP4/MOV/WebM file	User seeks to 5.234s
↓	↓
`avformat_open_input()`	Convert seconds → timebase units
↓	↓
Extract `RawMediaTimeSpan`:	`avformat_seek_file()`
`{ start, end, timebase }`	↓
	Position at prior keyframe (4.8s)

Key insight: Browsers have no demuxing API. libav.js provides container-level access that WebCodecs cannot.

Hybrid Architecture (2/2): WebCodecs for Decoding + Rendering

Browser APIs handle hardware-accelerated decode and display

Step 3: Decode Frames	Step 4: Render to Screen
Read packets via `av_read_frame()`	`VideoFrame` from decoder
↓	↓
Convert to `EncodedVideoChunk`	Upload to WebGL texture
↓	↓
`VideoDecoder.decode(chunk)`	Apply effects/compositing
↓	↓
Output: `VideoFrame`	Render to Canvas

Result: Fast seeking (libav.js knows keyframe locations) + hardware decode (WebCodecs)

Data Flow: libav.js Packets → WebCodecs

Converting WASM FFmpeg packets to browser API format

```
// Step 1: Read packet from libav.js (WASM FFmpeg)
const packetJs = await libav.ff_copyout_packet(this.packetPtr);
// Returns: { data: ArrayBuffer, pts: int64, flags: int }

// Step 2: Convert to WebCodecs EncodedVideoChunk
private createVideoChunk(packet: Packet): EncodedVideoChunk {
    const type = (packet.flags & 0x0001) ? 'key' : 'delta';
    const pts = this.libav.i64tof64(packet.pts ?? 0, packet.ptshi ?? 0);
    const timestamp = timestampToMicroseconds(pts, videoStream);

    return new EncodedVideoChunk({
        type,               // 'key' or 'delta'
        timestamp,          // microseconds
        data: packet.data,  // ArrayBuffer from WASM
    });
}

// Step 3: Pass to WebCodecs for hardware-accelerated decode
decoder.decode(chunk);
```

Listing 1: LibavDemuxer: Bridge from WASM to WebCodecs

Key point: ArrayBuffer from WASM memory → Browser API object

Media Transformation Server: Browser-to-Backend Handoff

When browser processing isn't enough:
offloading intensive operations to backend infrastructure

The Media Processing Challenge

Problem: User-provided media has heterogeneous characteristics

Raw Upload Characteristics

- **Diverse codecs/containers:** AV1, HEVC, VP9, ProRes, H.264 in MP4/MOV/MKV/WebM
- **Variable frame rate:** Keyframes 1-10s apart $\rightarrow$ slow seeking
- **Arbitrary resolution:** 4K/8K $\rightarrow$ inefficient for 1080p viewport
- **Heavy effects:** Background removal, eye contact exceed browser capabilities

Application Requirements

- **Fast seeking:** High keyframe rate (every 0.5-1s)
- **Bandwidth efficiency:** Match resolution to viewport
- **Heavy effects:** Server-side background removal, warping, color grading

Solution: Media Transformation Server (MTS)

MTS Timeline: Upload and Handoff

Seamless transition from local assets to server-processed media

1. **User adds media to composition**
   - Media file queued for upload to cloud storage
   - Immediate playback starts using local blob (instant feedback)
2. **While media is uploading**
   - Continue using local assets (see VFR seeking with libav.js)
   - Editor remains fully functional during upload
   - No user-visible interruption
3. **Once upload completes**
   - Client switches media references from local blob to MTS URLs
   - MTS begins processing: keyframe insertion, resolution scaling, effect application
   - Subsequent requests served from optimized, edit-ready media

Result: Zero-latency start (local), optimized playback (MTS), seamless transition

MTS: Technical Details and Trade-offs

Architecture

- **Thin wrapper above libav:** Rust FFI to FFmpeg for processing
- **Full MP4 segments (not HLS/fMP4):** Better browser compatibility
- **Dynamic segment generation:** Params (effects, resolution) change frequently
  - Generate on-demand with current parameters
  - Experimenting with pre-segmentation + caching for popular resolutions

Challenges

- **HDR & ProRes:** Wide color gamut → tone mapping, transcoding overhead
- **Latency:** First segment 500ms-2s → prefetching, caching strategies
- **User bandwidth:** 4K on slow connections → adaptive resolution
- **Feature availability:** Application functionalities depend on phase (local vs MTS) → some features unavailable until MTS processing begins

Open Questions

Looking ahead: Replicating browser compositions on backend infrastructure with perfect fidelity

Challenge: Browser-to-Backend Rendering Fidelity

Problem: Exporting final projects from browser to backend for rendering

Why Backend Rendering?

- Limited client CPU/GPU capacity, keep computer available during long exports
- Centralized rendering environment for consistency

Technical Challenges

- **Complexity:** Translating WebGL/WebGPU compositions to backend is non-trivial
- **Cross-platform fidelity:** Different GPU drivers, shader compilers, precision, color space, blending modes
- **State management:** Preserving all shader uniforms, transforms, effects

Goal: Pixel-perfect reproduction of browser composition on backend

Potential Solutions: Bridging the GL Gap

Emerging approaches for cross-platform graphics fidelity:

wgpu

- Mature (16k+ stars), used by Firefox & Bevy game engine
- Same code runs on Vulkan/Metal/DX12/OpenGL & WebGPU/WebGL2
- Headless rendering possible, WebGPU spec still evolving (Working Draft)

Other Approaches

- Headless GL libraries (headless-gl, webg3n) for server-side WebGL
- Google Dawn: Desktop WebGPU implementation abstracting backends

Open question: Can wgpu provide pixel-perfect fidelity across web & backend?

Questions?

Thank you!