CanvasForge Product Development

Analysis of Current State and Feasibility

Based on the provided project documentation, including the CHANGELOG.md and the latest code state in CanvasForge_251204_v1.md, CanvasForge is a mature PyQt6-based prototype focused on image editing with features such as selection overlays, context menus, flattening/saving workflows, and clipboard integration. The CHANGELOG indicates recent advancements in usability (e.g., hiding overlays before export, undo support, and artifact presets), positioning the application as a functional testbed for raster/vector manipulation. However, it lacks multimedia capture or narration capabilities, making the proposed demo tool an extension rather than a core evolution. Given your goals of refactoring to a low-level language (e.g., Rust or C++) for performance and modularity—drawing inspiration from Graphite.rs (a Rust-based procedural graphics editor) and Inkscape (C++-driven vector tool)—this feature should be prototyped in Python first to validate workflows, then refactored with interoperability in mind (e.g., via foreign function interfaces or modular crates/plugins).

The friend's idea aligns well as a "lightweight demo tool" for sharing app interactions, potentially integrated as a "Record Demo" mode within CanvasForge. It could leverage existing canvas/viewport logic for region selection and extend to capture changes, cursor tracking, and efficient export. Challenges include real-time change detection (requiring efficient diffing to minimize frames), audio generation (needing text-to-speech libraries), and sharing without a persistent server (e.g., via one-time URLs or peer-to-peer). From research using available tools, Python libraries like PyQt6 (for capture), MoviePy/FFmpeg (for encoding), gTTS/ElevenLabs (for narration), and Flask (for a minimal viewer) provide viable starting points. Refactoring to Rust/C++ would involve equivalents like winit/servo for UI, ffmpeg-rs for encoding, and actix-web for viewers, ensuring modules (e.g., capture logic) can be exposed as libraries for integration into tools like Graphite.rs.

Updated TODO.md

Below is the revised TODO.md with the new feature added under a dedicated "Medium Priority" section for balance (as it enhances usability but is not core to editing). The entry includes a high-level summary and cross-references to your goals for refactoring and integration.

```
# TODO

## High Priority

- **Shape stretching:** Add edge handles that scale items non-uniformly (cor
ners remain uniform) without degrading raster quality; selection overlay math
must update bounding boxes accordingly.
- **Fill tool + eyedropper:** Introduce a fill mode with a zoomed preview eyed
ropper, allowing users to sample on-canvas colors and fill selection cutouts or
shapes.

## Medium Priority

- **Lightweight Demo Recording Tool:** Implement a mode to capture selecte
d screen regions as freeze-frames on changes (~10 frames for short demos),
record cursor positions, generate AI-narrated audio/transcripts from context,
encode efficiently (e.g., H.265-like with timestamps), create shareable one-ti
me links, and provide a minimal web viewer. Prototype in Python to test workfl
ows; refactor to Rust/C++ for performance, ensuring modular design for integ
ration into editors like Graphite.rs or Inkscape (e.g., as a plugin or FFI-expose
d library).

## Planned Enhancements

- **Color/asset palettes:** Persist frequently used colors, callouts, and text st
yles for faster reuse across sessions.
- **Better export management:** Support manual filename entry, recent-save
history, and optional JPEG/SVG exports.
- **Undo/redo polish:** Expand undo granularity for selection overlays, flatten
operations, and save-directory changes.
- **Packaging:** Provide a platform-specific bundle (AppImage/Windows inst
```

aller) so users can run CanvasForge without a dev environment.
- **SVG editing & morphing:** Provide on-canvas vector node editing, path c ombination, and shape-morphing tools so pasted SVGs stay fully editable.
- **SVG template library:** Let users pin favorite SVG snippets/templates to a palette for single-click insertion.
- **Undo/redo polish:** Expand undo granularity for selection overlays, flatten operations, and save-directory changes.
- **Mojo compilation experiment:** Investigate compiling core routines with M odular's Mojo to evaluate performance gains.

Suggested Course of Action

To transition from the current prototype to this feature, follow a structured, iterative approach that leverages Python for rapid validation while planning for low-level refactoring. This ensures the demo tool remains lightweight (target: <10MB files, minimal dependencies) and modular (e.g., core capture/encoding as a separate module for easy porting). Estimated timeline: 2-4 weeks for Python prototype; 4-6 weeks for Rust/C++ refactor, assuming part-time development.

Phase 1: Research and Planning (1-2 days)

- **Review Dependencies:** Use Python libraries identified from research: PyQt6/QScreenCapture for region selection and frame grabs; scikit-image/Numpy for change detection (e.g., pixel diffs to trigger frames); pynput/MouseInfo for cursor tracking; gTTS or ElevenLabs API for audio narration (generate transcript first via context analysis or simple templating); MoviePy/FFmpeg-python for H.265 encoding (store frames + durations/timestamps in a container); Flask or http.server for a local web viewer (HTML5 video with controls); and services like Transfer.sh or self-hosted MinIO for one-time links (avoid persistent servers by using expiring uploads).
- **Design Modularity:** Structure as a CanvasView extension (e.g., "Record Mode" button) to reuse existing region selection. Define interfaces (e.g., abstract classes for capture/encode) that map to Rust crates (e.g., image for diffs, ffmpeg-next for encoding) or C++ libs (e.g., Qt's QScreenCapture, OpenCV for diffs).

- **Integration Considerations:** Ensure the module exports JSON specs (e.g., frame metadata) for easy import into Graphite.rs (Rust) or Inkscape (C++ via plugins). Use PyO3 for Python-Rust bindings during transition.

Phase 2: Python Prototype Implementation (1-2 weeks)

- **Step 1: Region Selection and Capture (3-5 days):** Extend CanvasView with a "Record Demo" action. Use QScreenCapture to grab initial region; implement a timer (QTimer) for periodic sampling (~100ms); detect changes via image diff (e.g., structural similarity index > threshold to add frame). Log cursor positions per frame.
- **Step 2: Narration Generation (2-3 days):** Analyze captured frames/context (e.g., OCR via Tesseract if needed) to generate transcript; use gTTS for audio. Sync audio timestamps with frame durations.
- **Step 3: Efficient Encoding and Saving (2-3 days):** Use MoviePy to compile frames + audio into H.265 (via FFmpeg backend); optimize with low CRF for small files. Auto-timestamp filenames (e.g., demo_YYYYMMDD_HHMM.mp4).
- **Step 4: Sharing and Viewer (2-3 days):** Upload to a service like AnonFiles/Transfer.sh for one-time links (copy to clipboard); create a simple Flask app for local viewing (HTML with <video> tag, optional frame-stepping).
- **Testing:** Verify on sample demos (e.g., app interactions); ensure file size <5MB for 2-min clips. Log issues to pasted_logs for debugging.

Phase 3: Refactoring to Low-Level Language (2-4 weeks)

- **Choose Target:** Rust for Graphite.rs-like performance (use winit for UI, image crate for processing); C++ for Inkscape integration (Qt base aligns with PyQt6).
- **Modular Breakdown:** Port capture as a standalone lib (Rust: screencapture-rs; C++: QtMultimedia). Use FFI (e.g., cxx for Rust-C++) for cross-editor compatibility.
- **Step 1: Prototype Port (1 week):** Rewrite core logic (capture/diff/encode) in Rust/C++; test via CLI before UI integration.

- **Step 2: Integration (1-2 weeks):** Embed as a plugin (e.g., Graphite.rs node graph for demo export); ensure API compatibility (e.g., expose C FFI for Inkscape).
- **Step 3: Optimization and Testing (1 week):** Benchmark (e.g., Rust's rayon for parallel diffs); add undo for recordings.

This course minimizes risk by validating in Python first, then refactoring for speed/modularity, aligning with your prototype-to-production vision. If additional research or prototyping is needed, I can refine further based on specifics.