Chrome Extension Workplan: Image Text Extractor

This plan outlines the steps to build a Chrome Extension that allows users to extract text from images or screen regions on any website and copy it to the Windows clipboard.

Technologies & Tools

To achieve this, we will use the following stack. Here is how they map to our development phases:

1. **Chrome Extension Manifest V3**: The core configuration standard for modern extensions.
    - *Usage*: Defined in **Phase 1 & 2** to set permissions and structure.
2. **JavaScript (ES6+)**: The primary language for logic.
    - *Usage*: Used throughout, specifically in `background.js` (logic) and `content.js` (page interaction).
3. **Tesseract.js**: A powerful JS library for Optical Character Recognition (OCR).
    - *Usage*: Integrated in **Phase 4**, used in **Phase 5 & 6** to process image data.
4. **Chrome APIs**:
    - `contextMenus` : For the right-click menu (**Phase 3**).
    - `tabs.captureVisibleTab` : To take screenshots (**Phase 6**).
    - `scripting` : To inject the selection overlay (**Phase 6**).
    - `runtime` : For messaging between background and content scripts (**Phase 5**).
5. **HTML5 Canvas**: For image manipulation.
    - *Usage*: Used in **Phase 6** to crop the screenshot to the user's selected area.
6. **Clipboard API**: Standard Web API.
    - *Usage*: Used in **Phase 7** to save the result.

Phase 1: Project Initialization

- [ ] **Create Project Structure**:
    - Create a root folder for the extension.
    - Create subfolders: `icons` , `lib` (for third-party libraries like Tesseract.js), `scripts` .
- [ ] **Manifest File**:
    - Create `manifest.json` using Manifest V3 standards.
    - Define metadata: Name, Version, Description.

Phase 2: Permissions & Configuration

- [ ] **Configure `manifest.json` Permissions**:
  - `"activeTab"` : To interact with the current tab.
  - `"scripting"` : To inject code for selection overlays.
  - `"contextMenus"` : To add right-click options on images.
  - `"clipboardWrite"` : To write the extracted text to the clipboard.
- [ ] **Host Permissions**:
  - Add `"<all_urls>"` to ensure the extension works on any website.

Phase 3: User Interface & Interaction

- [ ] **Context Menu**:
  - Create a `background.js` (Service Worker).
  - Use `chrome.contextMenus.create` to add:
    1. "Extract Text from Image" (Context: `image`).
    2. "Select Area to Extract" (Context: `page` , `selection`).
- [ ] **Popup (Optional)**:
  - Create `popup.html` and `popup.js` for settings (e.g., selecting OCR language).

Phase 4: OCR Integration (Tesseract.js)

- [ ] **Library Setup**:
  - Download `tesseract.min.js` and `worker.min.js` to the `lib` folder (local loading is preferred for extensions).
- [ ] **Content Script Injection**:
  - Create `content.js` to handle the actual processing on the web page.
  - Ensure Tesseract can be loaded within the content script context.

Phase 5: Implementation - Feature A (Right-click Image)

- [ ] **Event Handling**:
  - In `background.js` , listen for the context menu click.
  - Send a message to the active tab's `content.js` with the `srcUrl` of the image.
- [ ] **Processing**:

    - In `content.js`, receive the URL.
    - Convert the image URL to a format Tesseract accepts (Blob or Canvas if needed to avoid CORS issues).
    - Run `Tesseract.recognize()`.

Phase 6: Implementation - Feature B (Select Area/Crop)

- [ ] **Overlay UI**:
    - Create a script to inject a semi-transparent overlay on the page.
    - Implement mouse drag events to draw a selection rectangle.
- [ ] **Capture**:
    - Calculate the coordinates and dimensions of the selection.
    - Use `chrome.tabs.captureVisibleTab` in `background.js` to take a screenshot of the viewport.
    - Send the screenshot (Data URI) and coordinates to `content.js`.
- [ ] **Cropping**:
    - In `content.js`, create a hidden HTML5 Canvas.
    - Draw the screenshot onto the canvas.
    - Crop the canvas data to the selected coordinates.
    - Pass the cropped image data to Tesseract.

Phase 7: Clipboard & Feedback

- [ ] **Clipboard Action**:
    - Upon successful OCR, use `navigator.clipboard.writeText(result.text)`.
- [ ] **User Feedback**:
    - Implement a simple "Toast" notification (a small popup div injected into the page) to say "Text copied to clipboard!" or "Processing...".

Phase 8: Testing & Debugging

- [ ] **Load Extension**:
    - Go to `chrome://extensions`, enable Developer Mode, and "Load Unpacked".
- [ ] **Test Cases**:
    - Standard `<img>` tags.
    - Background images (via Area Selection).
    - Text inside videos (via Area Selection).
    - Cross-origin images (CORS handling).

Phase 9: Optimization

- [ ] **Performance**: Ensure Tesseract workers are terminated after use to save memory.
- [ ] **Accuracy**: Allow users to select languages if they copy non-English text.