Accessing Elements in HTML and Canvas Using Playwright

1. Accessing DOM Elements in HTML Using Playwright

Playwright is a powerful tool used to automate browsers like Chrome, Firefox, and Safari. It helps us simulate user actions like clicking buttons, typing in fields, and navigating web pages.

1. **Click a button by role and name:**

   await page.getByRole('button', { name: 'Submit' }).click();

   Clicks a button with accessible role button and visible name "Submit".

2. **Fill a text input by label:**

   await page.getByLabel('Username').fill('myUsername');

   Fills the input associated with the label "Username".

3. **Locate by test id:**

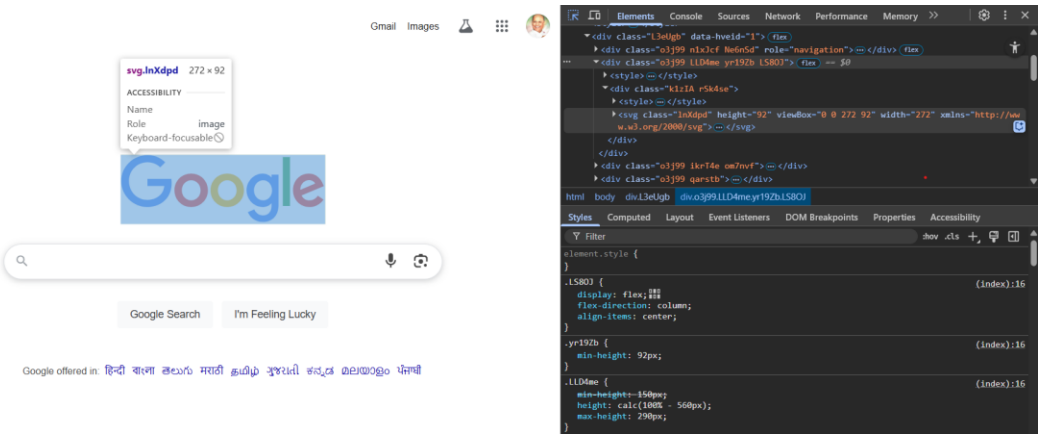
   await page.getByTestId(‘directions’).click();

   Finds and clicks a button with id submit.

4. **Type into an input by placeholder:**

   await page.getByPlaceholder('Enter your email').type(‘ab@gmail.com');

   Types into an input with the placeholder text "Enter your email".



2. Accessing Elements in Canvas

Unlike HTML elements, elements drawn on a <canvas> are not part of the DOM. The canvas is simply a bitmap that is rendered dynamically using JavaScript drawing APIs.
As a result, Playwright or any DOM-based automation tool cannot access or interact with individual canvas-drawn objects directly.

Challenges:

- Canvas does not expose individual elements as nodes in the DOM.
- There's no way to select or inspect elements using CSS or XPath.
- Canvas renders pixels, not structured data.

Solution: Template Matching with OpenCV

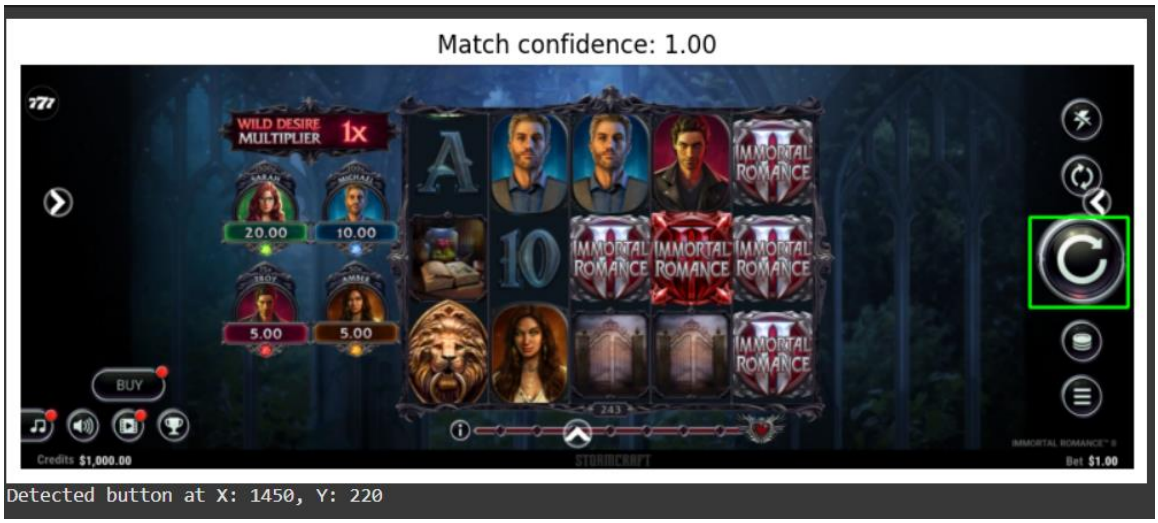
To overcome this limitation, we use image processing techniques like Template Matching with OpenCV. Here's how it works:

1. Take a screenshot of the canvas (using Playwright).
2. Use a reference image (template) of the UI element to detect (e.g., 'BUY' button).
3. Use OpenCV's template matching algorithm to find the coordinates of the element in the screenshot.
4. Use Playwright to simulate a mouse click at those coordinates.

Code:

```
import cv2

import numpy as np

from matplotlib import pyplot as plt

# Load main image and template

main_image = cv2.imread('/content/Photo1.jpg')

template = cv2.imread('/content/Photo3.jpg')

# Convert to grayscale

main_gray = cv2.cvtColor(main_image, cv2.COLOR_BGR2GRAY)
```

```
template_gray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)

# Get width and height of the template

w, h = template_gray.shape[::-1]

# Apply template matching

result = cv2.matchTemplate(main_gray, template_gray, cv2.TM_CCOEFF_NORMED)

# Get best match position

min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)

# Draw rectangle around matched region

top_left = max_loc

bottom_right = (top_left[0] + w, top_left[1] + h)

cv2.rectangle(main_image, top_left, bottom_right, (0, 255, 0), 3)

# Show result

plt.figure(figsize=(10, 6))

plt.imshow(cv2.cvtColor(main_image, cv2.COLOR_BGR2RGB))

plt.title(f"Match confidence: {max_val:.2f}")

plt.axis('off')

plt.show()

# Print coordinates

print(f"Detected button at X: {top_left[0]}, Y: {top_left[1]}")
```



Then click using Playwright at detected coordinates:

For e.g. -> await page.mouse.click(101, 427);

Advantages of This Approach:

- Works with canvas-based games where DOM is not available.
- Detects UI elements visually with high confidence (e.g., Match confidence: 1.00).
- Easily adaptable to different screen resolutions.

Colab:

https://colab.research.google.com/drive/15O2xsxYZTf9l5YNvIbbHbSUMP51XfHUz?authuser=0#scrollTo=NRqMJ2RW7xNH