PixelWeb: The First Web GUI Dataset with Pixel-Wise Labels

Qi Yang
Peking University
Beijing, China
qi.yang@stu.pku.edu.cn

Weichen Bi
Peking University
Beijing, China
biweichen@pku.edu.cn

Haiyang Shen
Peking University
Beijing, China
hyshen@stu.pku.edu.cn

Yaoqi Guo
Peking University
Beijing, China
ianwalls@pku.edu.cn

Yun Ma
Peking University
Beijing, China
mayun@pku.edu.cn

Abstract

Graphical User Interface (GUI) datasets are crucial for various downstream tasks. However, GUI datasets often generate annotation information through automatic labeling, which commonly results in inaccurate GUI element BBox annotations, including missing, duplicate, or meaningless BBoxes. These issues can degrade the performance of models trained on these datasets, limiting their effectiveness in real-world applications. Additionally, existing GUI datasets only provide BBox annotations visually, which restricts the development of visually related GUI downstream tasks. To address these issues, we introduce PixelWeb, a large-scale GUI dataset containing over 100,000 annotated web pages. PixelWeb is constructed using a novel automatic annotation approach that integrates visual feature extraction and Document Object Model (DOM) structure analysis through two core modules: channel derivation and layer analysis. Channel derivation ensures accurate localization of GUI elements in cases of occlusion and overlapping elements by extracting BGRA four-channel bitmap annotations. Layer analysis uses the DOM to determine the visibility and stacking order of elements, providing precise BBox annotations. Additionally, PixelWeb includes comprehensive metadata such as element images, contours, and mask annotations. Manual verification by three independent annotators confirms the high quality and accuracy of PixelWeb annotations. Experimental results on GUI element detection tasks show that PixelWeb achieves performance on the mAP95 metric that is 3-7 times better than existing datasets. We believe that PixelWeb has great potential for performance improvement in downstream tasks such as GUI generation and automated user interaction.

Keywords

GUI dataset; Web page; Automated annotation

1 Introduction

The increasing complexity and diversity of graphical user interfaces (GUIs) of web applications underscore the necessity for more precised GUI modeling, which is essential for various downstream tasks, such as GUI code generation [19, 21], GUI retrieval [5], and user interaction automation through LLM-based agents [14, 18, 23].

Precise GUI modeling relies on high-quality and large-scale datasets for model training. However, existing widely-used GUI datasets, such as WebUI [20], Rico [7], and MUD [9], suffer from pervasive annotation inaccuracies [13]. Figure 1 shows some cases of imprecise BBox labels in WebUI dataset: (a) some BBoxes do not have corresponding GUI elements; (b) the same GUI element corresponds to multiple BBoxes; (c) the positions of some BBoxes do not align with the GUI elements. These annotation issues can introduce noise and undermine the reliability of trained models, adversely affecting the performance of downstream tasks.

Addressing these limitations faces two primary challenges: (1) Unknown Accurate Coordinates of Elements: Existing browser APIs for obtaining element coordinates often fail to accurately reflect the actual positions and sizes of GUI elements [13], especially for non-rectangular and dynamically rendered components. This leads to incorrect BBox sizes and imprecise localization; (2) Unknown Visibility of Elements: Determining the visibility of elements is complicated by factors such as overlapping components and varying display conditions [13]. Simple code analysis cannot reliably determine whether elements are visible to users, resulting in missing, duplicate, or meaningless BBoxes.

To this end, we propose an automated annotation approach designed to enhance the accuracy of GUI dataset construction. Our approach leverages the synergy between visual features and DOM structure through two core modules: *Channel Derivation* and *Layer Analysis*. Channel Derivation extracts the BGRA channels for each pixel of GUI elements, ensuring accurate localization even in complex scenarios involving occlusions and overlapping elements. Layer Analysis examines the element hierarchy by analyzing the DOM tree and z-index information, determining the visibility and stacking order of elements. Together, these modules generate precise BBox annotations and comprehensive metadata with minimal human intervention.

Based on this approach, we construct ***PixelWeb*** [1], a Web GUI dataset comprising **100,000** annotated web pages. PixelWeb offers more precise BBox annotations and more metadata labels of each element such as image, mask, and contour. Experimental results demonstrate that models trained on PixelWeb significantly outperform those trained on existing datasets, highlighting the efficacy of our approach in producing high-quality annotations.

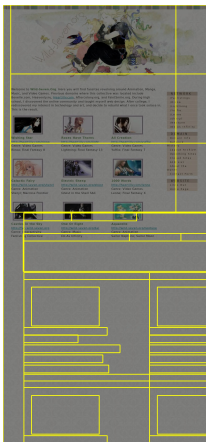
Experimental evaluations demonstrate the efficacy of approach in producing high-quality annotations. Manual verification conducted by three independent annotators revealed higher annotation quality for BBox annotations and more comprehensive metadata. Furthermore, experimental results indicate that element detection models built using PixelWeb significantly outperform existing GUI datasets in detection accuracy. This implies that PixelWeb can play

[1] PixelWeb can be accessed from https://huggingface.co/datasets/cyberalchemist/PixelWeb

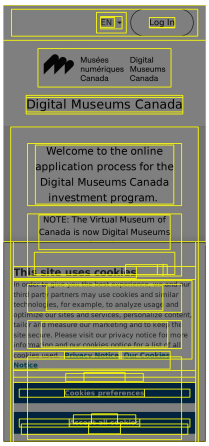
a greater role in more downstream tasks based on element detection, such as GUI operations and understanding.

In summary, our contributions are as follows.

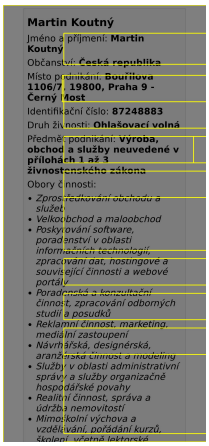
- **Pixel-wise Annotation Approach for Web GUI**: We propose an automated annotation approach that integrate visual features and DOM structure to overcome existing annotation challenges.
- **PixelWeb Dataset**: We construct a large-scale, automated GUI dataset encompassing 100,000 web pages with more precise BBox annotations, as well as more metadata labels, such as the image, mask and contour of each element.
- **Enhanced Performance of Downstream Tasks**: We validate the effectiveness of PixelWeb through extensive experiments, demonstrating significant improvements in GUI element detection, thereby underscoring the dataset's value for more downstream tasks.



(a) The positions of the BBoxes do not have GUI elements



(b) The same element corresponds to multiple BBoxes.



(c) The positions of the BBoxes are offset relative to the GUI elements.

Figure 1: Error cases of BBox label in WebUI dataset

2 Related Work

In this section, we survey related work on GUI datasets and downstream tasks enabled by GUI datasets.

2.1 GUI Datasets

The construction of GUI datasets has been extensively explored to support downstream tasks in UI analysis and automation. Some research improves downstream task performance by constructing datasets containing diverse GUI-related information. For instance, Rico [7] is a seminal mobile UI repository constructed by combining crowdsourcing and automation to extract visual, structural, and interactive properties from thousands of apps. VINS [5] is a dataset with hierarchical UI structure annotations to enable object-detection-based retrieval. UICrit [8] focuses on enhancing automated UI evaluation by curating a critique dataset to refine LLM-generated feedback, bridging the gap between automated and human evaluators. WebUI [20] crawls web pages to create a large-scale dataset with web semantics for cross-domain visual UI understanding. Some work focuses on automating dataset acquisition to reduce manual hard labor. For example, CLAY [13] proposes a deep learning pipeline to denoise raw mobile UI layouts, automating dataset refinement and reducing manual labeling efforts. MUD [9] employs LLMs to mine modern UIs from apps, integrating noise filtering and human validation to address outdated or noisy data. However, previous work faced inherent limitations in annotation precision (e.g., noisy BBoxes, overlapping elements) or scalability due to manual interventions. Our approach proposes pixel-level precise annotations, which helps further enhance the performance of downstream tasks.

Table 1: Annotation differences among datasets

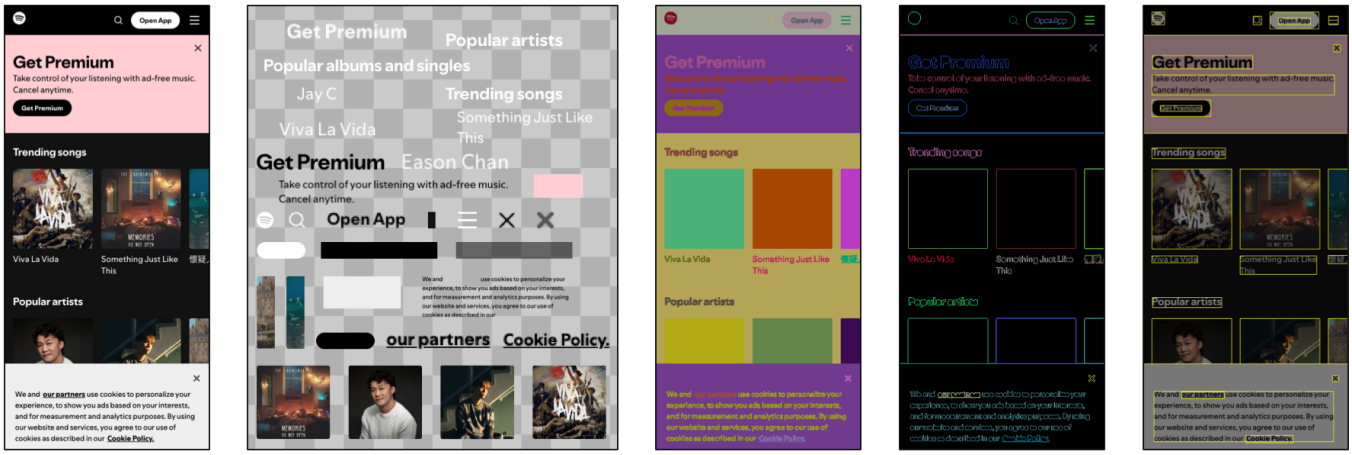
Features	PixelWeb	WebUI [20]	Rico [7]	MUD [9]
Page Screenshot	✓	✓	✓	✓
Page Hierarchy Code	✓	✓	✓	✓
Page Animation			✓	
Element Image	✓			
Element Layer	✓			
Element Mask	✓			
Element Contour	✓			
Element BBox	✓	✓	✓	✓
Element Class	✓	✓		✓
Element Computed Style	✓			

2.2 GUI Tasks

Numerous studies have focused on advancing performance in GUI-related tasks. Some work focuses on GUI element detection: UIED [22] combines traditional CV and deep learning for element detection, and Chen et al. [6] merge coarse-to-fine strategies with deep learning. Another task is GUI retrieval. For instance, Guigle [4] helps conceptualize the user interfaces of the app by indexing GUI images and metadata for intuitive search. In layout generation, approaches like BLT [12] and LayoutDM [11] improve controllability and quality, while LayoutTransformer [10] unifies cross-domain layout synthesis. Furthermore, GUI agents such as AutoGLM [15] integrate reinforcement learning for autonomous control. Compared to previous work, our approach leverages a pixel-level annotated dataset (PixelWeb), enabling enhanced performance across GUI tasks.

3 Approach

In this section, we present an automated annotation approach for Web GUIs that enables large-scale dataset construction, which delivers (1) relatively precise BBox annotations and (2) more metadata labels, including the image, mask, contour of each GUI element. The approach consists of two core modules: (1) *Channel Derivation* (Section 3.2), which precisely extracts the image of each GUI element, and (2) *Layer Analysis* (Section 3.3), which examines element hierarchy as well as produces lower-dimensional annotation (masks, contours, and BBoxes).



(a) GUI Screenshot (b) Extracted Images of Each GUI Element (c) Element Masks (d) Element Contours (e) Element BBoxes

Figure 2: Example of a web page annotated by our approach

3.1 Overview

Achieving precise automated annotation of GUI elements from web pages faces two technical challenges:

- **Unknown Accurate Coordinates of Elements**: Although browsers provide various APIs to obtain element coordinates, they cannot accurately reflect the actual positions of elements, leading to incorrect BBox sizes.
- **Unknown Visibility of Elements**: The visibility of elements is influenced by multiple factors, making it difficult to determine whether elements are visible to users through simple code analysis. This can result in missing BBoxes, duplicate BBoxes, and meaningless BBoxes.

To address these challenges, we designed two modules: channel derivation and layer analysis. An overview of our approach is shown in Figure 3, which consists of three steps:

(1) Open the target webpage to be annotated.
(2) Obtain the BGRA four-channel bitmap and XY coordinates of each GUI element through **Channel Derivation**.
(3) Determine the layer position (i.e., the Z-axis coordinate) of each GUI element through **Layer Analysis**.

Using the information obtained from these modules, we can further derive image annotation data such as masks, contours, and BBoxes.

Channel Derivation: By independently displaying the web elements to be annotated and changing the page's background color, a chroma key group can be obtained. This allows us to solve the color composition equations to obtain the BGRA four-channel bitmap of each web element.

Layer Analysis: Based on the webpage's XPath and z-index information, we construct vectors that describe the stacking order of overlapping elements on the page. This results in several directed acyclic graphs (DAGs), which are then analyzed using a topological sorting algorithm to determine the hierarchy of elements.

After processing with these two modules, we obtain the image and spatial coordinates of each element. Using this high-dimensional information, we can derive lower-dimensional annotation data. Table 1 illustrates the differences in annotations provided by our dataset compared to others. Our dataset offers more comprehensive information at the element level. Although we do not include the Animation annotations available in the Rico dataset, the Rico dataset requires manual annotation, and our targeted WebUI dataset also does not provide such annotations.

Figure 2 displays a webpage annotated using our approach. The primary distinction between our GUI dataset and previous ones is the additional *element images* label shown in Figure 2b. By combining this with the rendering hierarchy, we can compute annotation information such as the mask(2c), contour2d, and BBox(2e) of elements.

3.2 Channel Derivation

Extracting GUI element images directly from GUI screenshots is a challenging task due to various special cases that make the direct use of simple cropping and contour extraction algorithms infeasible. For instance, elements may be occluded, leading to some regions being invisible. Additionally, the boundaries between elements may be blurred, making it difficult to accurately delineate areas. Furthermore, some elements may have the same color as the background, making it challenging to distinguish between the elements themselves and the page background.

Therefore, we designed a approach to extract GUI element images as follows: First, hide all elements except the target element, then change the background color of the HTML page to obtain screenshots with different chroma keys. This allows us to derive the BGRA color channel values of any pixel of the target element. We specifically designed a color derivation algorithm for this purpose. Using the standard color composition formula1:

$$V_{\text{rendered}} = \alpha \times V_{\text{foreground}} + (1 - \alpha) \times V_{\text{background}} \quad (1)$$

we can obtain GUI screenshots under different background overlays. Through these screenshots and the color composition formula,

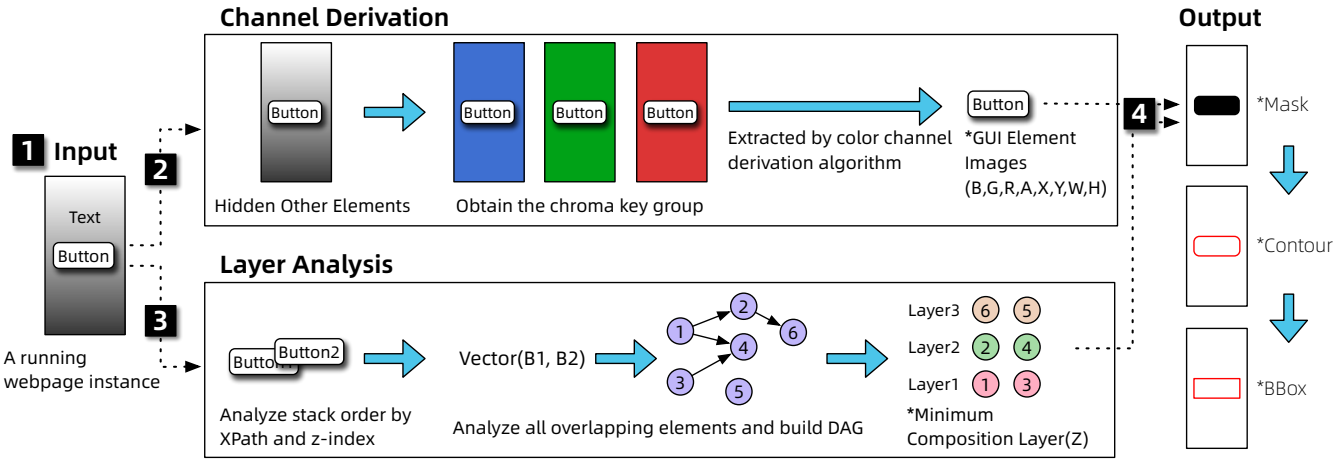


Figure 3: Approach overview. 1. Input an open web page. 2. The channel derivation module extracts the image of each element. 3. The layer analysis analyzes the rendering layer of elements to determine the image and coordinates of each GUI element. 4. Based on this information, it sequentially derives mask, contour, and BBox annotations.

we deduce the BGRA channel values of each pixel of the foreground GUI element.

Due to the characteristics of HTML syntax, a hierarchical tree node may contain more than one visual object. Although we can hide or display any HTML node, it is not possible to simply set a node's text to be visible while keeping the background hidden. This prevents us from obtaining a text chroma key group by overlaying backgrounds of different colors. Therefore, we designed two extraction approach, applicable to situations where the background can be removed and where it cannot. For cases where the background can be removed, we perform a color transformation on the background to obtain the chroma key group. In situations where the background cannot be removed, we perform a color transformation on the foreground to obtain the chroma key group. In the following text, we refer to the former image as a graphic and the latter as text.

We designed different processing workflows for the above two situations, as shown in Figure 4. Regardless of the situation, we first hide all other irrelevant elements, retaining only the background of the HTML itself 4a. For graphic extraction 4b, we independently display it and change the background color to (255,0,0), (0,255,0), (0,0,255) to obtain the chroma key group. We solve for the BGRA channel values using the corresponding color channel derivation formula. For text extraction 4c, we capture a screenshot retaining the element and one without the element to obtain the element's mask. We then change its own color and use channel derivation to obtain its BGRA channels.

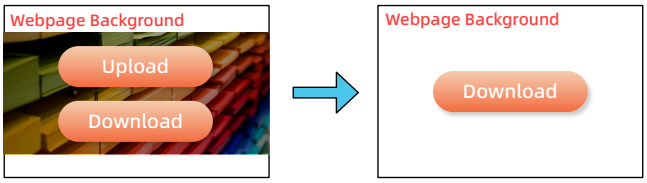
In the detailed description of the formulas that follow, we refer to the object to be extracted from the GUI screenshot as the foreground, and its background as the background. On this point, we consider all content other than the target to be extracted as the background, even if there are multiple layers, treating them as a single background layer. Whether extracting graphics or text, in their respective chroma key groups, they belong to the foreground, and anything other than themselves is considered the background.

Table 2: List of variables and descriptions

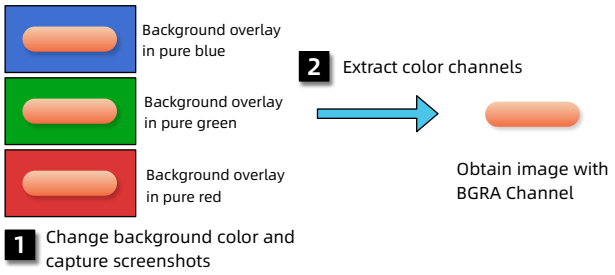
Variable	Description
ϵ	A very small value to prevent division by zero
$(B, G, R, A)_{\text{FG}}$	Graphic color channels
$(B, G, R, A)'_{\text{FG}}$	Text color channels
$(B, G, R)_{\text{BG}}$	Graphic background color channels
$(B, G, R)'_{\text{BG}}$	Text background color channels
$(B, G, R)_{\text{BlueBG}}$	Color channel values of GUI screenshot with blue background overlay
$(B, G, R)_{\text{GreenBG}}$	Color channel values of GUI screenshot with green background overlay
$(B, G, R)_{\text{RedBG}}$	Color channel values of GUI screenshot with red background overlay
$A'_{\text{(Blue,Green,Red)}}$	Alpha channel values calculated from different color channels
$Mask_{\text{WhiteBG}}$	Area where the background is pure white
$(B, G, R)_{\text{BlueFG}}$	Color channel values of a GUI screenshot when the text is set to blue
$(B, G, R)_{\text{GreenFG}}$	Color channel values of a GUI screenshot when the text is set to green
$(B, G, R)_{\text{RedFG}}$	Color channel values of a GUI screenshot when the text is set to red
$(B, G, R)_{\text{RawFG}}$	Color channel values of a GUI screenshot when the text is keep unchanged

For the range of color channels, we use the standard [0,255] for the B, G, R channels, and [0,1] for the alpha channel.

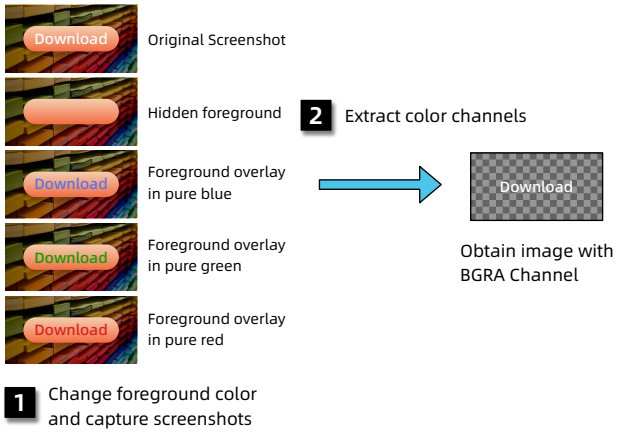
Figures 5 and 6 are examples of extreme cases of the bgra channel derivation formula and show the inputs and outputs of the formula. We use different extreme colors to test the formula, including pure white, pure black, gradient gray, pure red, pure green, and pure blue. Pure white and pure black represent the maximum and minimum



(a) Exclude interfering elements. Hide all elements except the target elements to be extracted



(b) Extract graphic color channel: The three GUI screenshots on the left from top to bottom are: background set to blue, background set to green, background set to red

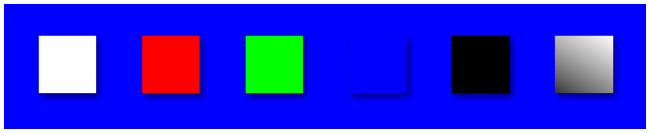


(c) Extract text color channel: The five GUI screenshots on the left from top to bottom are: no changes, text removed, text set to blue, text set to green, text set to red. Because the font is white, we overlay a checkerboard pattern background for better visualization.

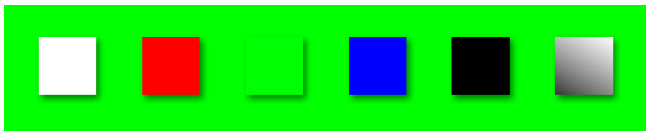
Figure 4: Channel derivation

values of all color channels, respectively; pure red, pure green, and pure blue represent the maximum values of each color channel in the RGB color space; gradient gray is used to test the formula's ability to handle mid-tone pixels.

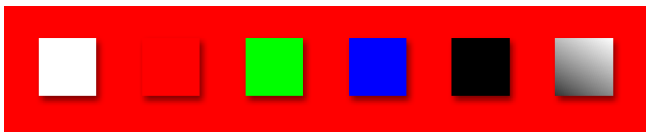
Figure 5 shows examples of graphic extraction. Figure 5a shows a screenshot when the GUI background is set to pure blue, Figure 5b shows a screenshot when the GUI background is set to pure green, Figure 5c shows a screenshot when the GUI background is set to



(a) Input 1: Blue background



(b) Input 2: Red background



(c) Input 3: Red background

(d) Output: BGRA channels of foreground

Figure 5: Examples of extreme cases for graphic color channel derivation formula

pure red, and Figure 5d shows the output obtained through the graphic bgra channel derivation formula from Figures 5a, 5b, 5c.

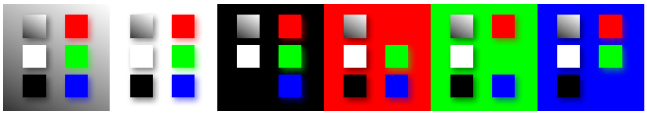
Figure 6 shows examples of text extraction. Figure 6a shows the original GUI screenshot, Figure 6b shows the GUI screenshot after hiding the text, Figure 6c shows a screenshot when the text is set to pure blue, Figure 6d shows a screenshot when the text is set to pure green, Figure 6e shows a screenshot when the text is set to pure red, and Figure 6f shows the output obtained through the text bgra channel derivation formula from Figures 6a, 6b, 6c, 6d, 6e.

First, we will introduce the extraction of graphics. Using the standard formula for color composition 1, we can list the system of equations shown in formula 2.

$$\begin{cases} (B,G,R)_{\text{BlueBG}} = A_{\text{FG}} \cdot (B,G,R)_{\text{FG}} + (1 - A_{\text{FG}}) \cdot (255,0,0) \\ (B,G,R)_{\text{GreenBG}} = A_{\text{FG}} \cdot (B,G,R)_{\text{FG}} + (1 - A_{\text{FG}}) \cdot (0,255,0) \\ (B,G,R)_{\text{RedBG}} = A_{\text{FG}} \cdot (B,G,R)_{\text{FG}} + (1 - A_{\text{FG}}) \cdot (0,0,255) \end{cases} \tag{2}$$

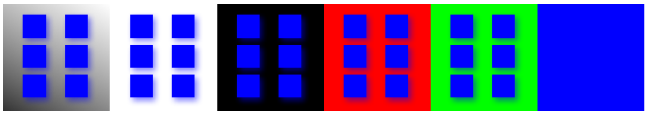
We can first solve for the value of the A channel of the graphic using the system of equations 2.

$$A_{\text{FG}} = 1 - \frac{1}{6}\left(\frac{|B_{\text{BlueBG}} - B_{\text{RedBG}}|}{255} + \frac{|B_{\text{BlueBG}} - B_{\text{GreenBG}}|}{255} + \frac{|G_{\text{GreenBG}} - G_{\text{BlueBG}}|}{255} + \frac{|G_{\text{GreenBG}} - G_{\text{RedBG}}|}{255} + \frac{|R_{\text{RedBG}} - R_{\text{GreenBG}}|}{255} + \frac{|R_{\text{RedBG}} - R_{\text{BlueBG}}|}{255} \right) \tag{3}$$

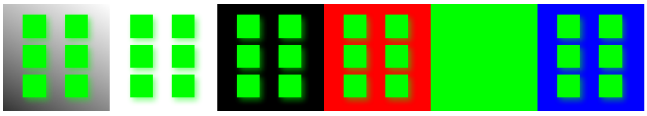


(a) Input 1: Original GUI Screenshot

(b) Input 2: Hidden foreground



(c) Input 3: Blue foreground



(d) Input 4: Green foreground

(e) Input 5: Red foreground

(f) Output: BGRA channels of foreground

Figure 6: Examples of extreme cases for text color channel derivation formula

Finally, we can substitute the value of the A channel into formula 1 to obtain the expressions for the blue, green, and red channel values 4.

$$\begin{cases} B_{\text{FG}} = \dfrac{B_{\text{BlueBG}} - (1 - A_{\text{FG}}) \cdot 255}{A_{\text{FG}} + \epsilon} \\ G_{\text{FG}} = \dfrac{G_{\text{GreenBG}} - (1 - A_{\text{FG}}) \cdot 255}{A_{\text{FG}} + \epsilon} \\ R_{\text{FG}} = \dfrac{R_{\text{RedBG}} - (1 - A_{\text{FG}}) \cdot 255}{A_{\text{FG}} + \epsilon} \end{cases} \tag{4}$$

Next, we will detail the extraction of text. For the set of GUI screenshots obtained by setting different color values, solving for the BGRA of text elements becomes more challenging. This is because the background of text nodes can be any background, making both the background channel values and the foreground (text) channel values unknown. We can first list the equations as follows:

$$\begin{cases} (B,G,R)_{\text{BlueFG}} = A'_{\text{FG}} \cdot (B,G,R) + (1 - A'_{\text{FG}}) \cdot (B,G,R)'_{\text{BG}} \\ (B,G,R)_{\text{GreenFG}} = A'_{\text{FG}} \cdot (B,G,R) + (1 - A'_{\text{FG}}) \cdot (B,G,R)'_{\text{BG}} \\ (B,G,R)_{\text{RedFG}} = A'_{\text{FG}} \cdot (B,G,R) + (1 - A'_{\text{FG}}) \cdot (B,G,R)'_{\text{BG}} \end{cases} \tag{5}$$

Based on the system of equations 5, we can first solve the system of equations 6 that describes the foreground alpha channel:

$$\begin{cases} A'_{\text{Blue}} = \dfrac{B_{\text{BlueFG}} - B'_{\text{BG}}}{255 - B'_{\text{BG}} + \epsilon} \\ A'_{\text{Green}} = \dfrac{G_{\text{GreenFG}} - G'_{\text{BG}}}{255 - G'_{\text{BG}} + \epsilon} \\ A'_{\text{Red}} = \dfrac{R_{\text{RedFG}} - R'_{\text{BG}}}{255 - R'_{\text{BG}} + \epsilon} \end{cases} \tag{6}$$

Since we calculate alpha from three different color channels, and there is only one final alpha channel, we use equation 7 to obtain the final alpha channel value.

$$A'_{\text{FG}} = \max(A_B, A_G, A_R) \tag{7}$$

According to the system of equations 6, it can be observed that when the background color is pure white, i.e., the color channel value is 255, the resulting alpha channel value ranges from negative infinity to zero, making the above equations inapplicable. However, when the background color is pure white, the foreground color is easy to determine, and the color derivation formula can be simplified to equation 8:

$$A = (255 - ChannelValue)/255 \tag{8}$$

$$Mask_{\text{WhiteBG}} = (B_{\text{BG}} = 255) \wedge (G_{\text{BG}} = 255) \wedge (R_{\text{BG}} = 255) \tag{9}$$

Therefore, we designed a mask extraction approach to find all pixel positions with a pure white background by detecting whether the color channel values are $(255, 255, 255)$:

$$A'_{\text{FG}} = \frac{1}{6}\left(\frac{255 - G_{\text{BlueFG}}}{255} + \frac{255 - R_{\text{BlueFG}}}{255} + \frac{255 - B_{\text{GreenFG}}}{255} + \frac{255 - R_{\text{GreenFG}}}{255} + \frac{255 - B_{\text{RedFG}}}{255} + \frac{255 - G_{\text{RedFG}}}{255}\right) \tag{10}$$

Finally, we apply equation 10 to the pixels within the mask area and equation 7 to other areas to obtain the alpha channel for all pixels of the target element. With the alpha channel, we can use the following equations 11 to obtain the BGR channel values.

$$\begin{cases} B'_{\text{FG}} = \dfrac{B_{\text{RawFG}} - (1 - A'_{\text{FG}}) \cdot B'_{\text{BG}}}{A'_{\text{FG}} + \epsilon} \\ G'_{\text{FG}} = \dfrac{G_{\text{RawFG}} - (1 - A'_{\text{FG}}) \cdot G'_{\text{BG}}}{A'_{\text{FG}} + \epsilon} \\ R'_{\text{FG}} = \dfrac{R_{\text{RawFG}} - (1 - A'_{\text{FG}}) \cdot R'_{\text{BG}}}{A'_{\text{FG}} + \epsilon} \end{cases} \tag{11}$$

3.3 Layer Analysis

In this step, our objective is to obtain the minimum composition layer (*MCL*), which involves assigning all elements in a GUI to different layers such that there is no overlap among elements within each layer, and any element in an upper layer is rendered above elements in a lower layer. Given a set of GUI elements $E = \{e_1, e_2, \ldots, e_n\}$, each element e_i is associated with a mask region $M(e_i)$. We aim to allocate these elements into a set of layers $L = \{L_1, L_2, \ldots, L_k\}$, where k is the number of layers, and our goal is to minimize k. Equation (12) describes the constraints of this optimization problem: the union of elements across all layers should equal the entire set of GUI elements E, i.e., $\bigcup_{i=1}^{k} L_i = E$; for any layer L_i, the mask regions of any two elements e_j and e_l should not overlap, i.e., $M(e_j) \cap M(e_l) = \emptyset$; furthermore, for each pair of adjacent layers L_m and L_{m+1}, if elements e and e' have overlapping mask regions, then the rendering order of e' must be below that of e. Through these conditions, we define how to efficiently assign GUI elements to the minimum number of layers, thereby obtaining an equivalent Z-axis for the actual composite rendering of the GUI elements.

$$
\begin{aligned}
&\text{Minimize} \quad k \\
&\text{Subject to} \quad \bigcup_{i=1}^{k} L_i = E, \\
&\qquad \forall i, \forall e_j, e_l \in L_i,\ R(e_j) \cap R(e_l) = \emptyset, \\
&\qquad \forall m, \forall e \in L_{m+1}, \forall e' \in L_m, \\
&\qquad \text{if } R(e) \cap R(e') \neq \emptyset, \text{ then render order of } e > e'.
\end{aligned} \tag{12}
$$

We designed an algorithm to obtain the MCL as follows: First, based on the Channel Derivation from the previous step, we have obtained the XY coordinates of each GUI element, allowing us to determine which elements overlap. We then perform pairwise analysis on these overlapping elements. Our approach is inspired by the insight that if we know the stack order of any two elements, we can construct a Directed Acyclic Graph (DAG) and solve for the MCL through topological sorting.

Algorithm 1 Analysis stack order of two GUI elements

```
if zindex_i is an integer ∧ zindex_j is an integer then
    if zindex_i = zindex_j then
        if w_i × h_i > w_j × h_j then return (1, 0)
        else if w_i × h_i < w_j × h_j then return (0, 1)
        end if
    else if zindex_i < zindex_j then return (1, 0)
    elsereturn (0, 1)
    end if
else
    if w_i × h_i > w_j × h_j then return (1, 0)
    else if w_i × h_i < w_j × h_j then return (0, 1)
    end if
end if
```

We use binary vectors to represent the stacking order of any two GUI elements: 1. If element e_i is above element e_j, we use the vector $\mathbf{v}_{ij} = (1, 0)$ to represent this order. 2. Conversely, if element e_j is above element e_i, the vector is $\mathbf{v}_{ji} = (0, 1)$.

According to the rendering mechanism of HTML, descendant nodes are above the rendering level of parent nodes. Therefore, we only need to compare two overlapping GUI elements that are not on the same XPath. Code 1 shows the comparison of two GUI elements through Z-Index and area. First, we compare their Z-Index attributes. If both GUI elements have numeric Z-Index values, we determine the stacking order based on the size of the Z-Index. If the Z-Index attribute is meaningless, such as auto, or if the Z-Index values of the two GUI elements are the same, we determine their hierarchy by comparing the areas of the two GUI elements. Based on our experience, larger elements are usually below smaller elements.

For all elements on the page $E = \{e_1, e_2, \ldots, e_n\}$, we can construct all possible binary vectors to represent their stacking order. Additionally, for elements in the set E, if certain elements do not overlap with any other elements, these elements can be considered as isolated vertices. These isolated vertices have no edges connecting them to other nodes in the DAG, indicating that their stacking order is independent of other elements:

$$
V = \left\{ \mathbf{v}_{ij} \mid e_i, e_j \in E,\ i \neq j \right\} \cup \{ e_k \mid e_k \in E,\ e_k \text{is isolated vertex} \} \tag{13}
$$

Finally, we apply Kahn's Algorithm for topological sorting on the DAG represented by Equation 13 to obtain the MCL that minimizes k. Each layer of the MCL records all elements present in that layer.5

3.4 Output Annotations

We process each HTML node on the page through the channel derivation and layer analysis modules, obtaining the BGRA bitmap of each GUI element, as well as the accurate spatial coordinates (X, Y, Z) and size information (W, H) of the GUI elements. Based on this information, we derive the annotations for the Mask, Contour, and BBox.

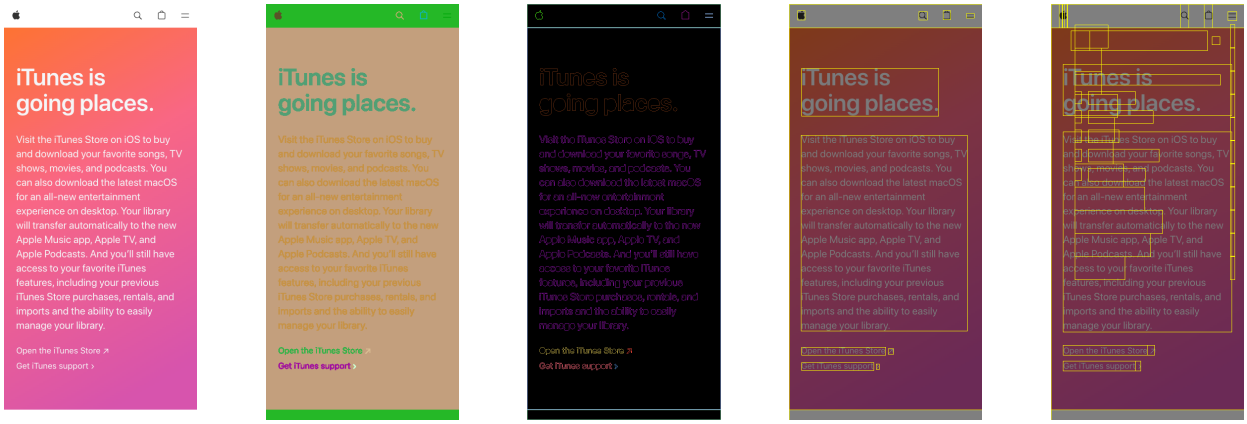
(1) **Mask**: First, define the BGRA bitmap of the target element as matrix M, where the α channel value of each pixel is $M(x, y, \alpha)$. We set all pixels satisfying $M(x, y, \alpha) \neq 0$ to color C_1, while pixels of other elements are set to color C_2, where C_2 applies to all non-target elements with non-zero α channel values.Next, recompose according to the layer order using Equation 1 to obtain a GUI page image, denoted as $\text{Element}_i^{\text{Visible}}$.Then, compose another GUI page image excluding the visible part of the target element, denoted as $\text{Element}_i^{\text{Invisible}}$.Finally, calculate the difference between the two images to get mask:

$$
D = \left| \text{Element}_i^{\text{Visible}} - \text{Element}_i^{\text{Invisible}} \right|
$$

D is the mask of the element's visible area.

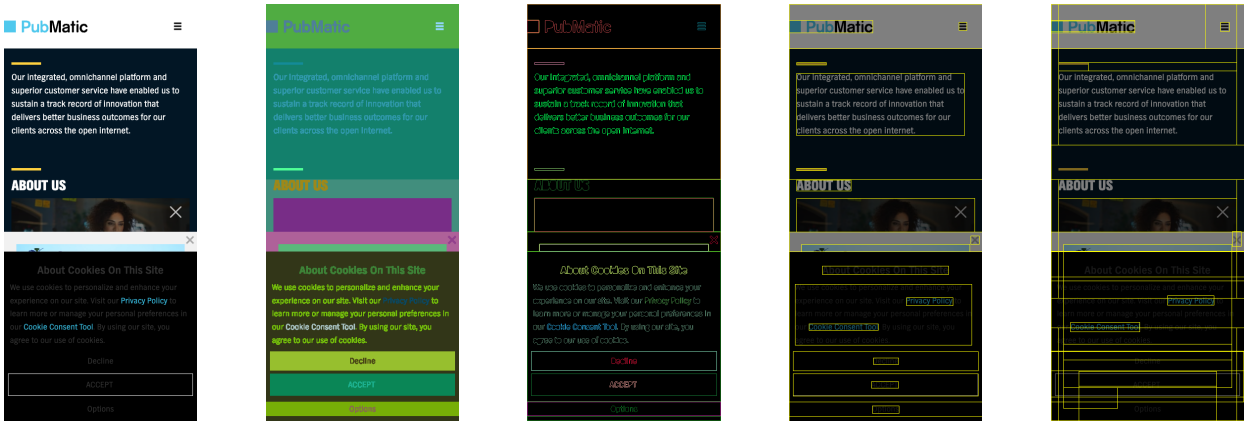
(2) **Contour**: We find the outer contour of all disconnected regions in the mask to obtain the contour.5

(3) **BBox**: We calculate the minimum enclosing rectangle of the mask to obtain the BBox.



(a) GUI screenshot (b) P-Web mask (c) P-Web contour (d) P-Web BBox (e) WebUI BBox

Figure 7: Example 1 of annotations in P-Web and WebUI



(a) GUI screenshot (b) P-Web Mask (c) P-Web Contour (d) P-Web BBox (e) WebUI BBox

Figure 8: Example 2 of annotations in P-Web and WebUI

(a) GUI screenshot (b) P-Web Mask (c) P-Web Contour (d) P-Web BBox (e) WebUI BBox

Figure 9: Example 3 of annotations in P-Web and WebUI

4 Implementation

The implementation of our approach primarily includes three components: page crawler, page operation, and annotation export.

The page crawler component is implemented based on package playwright [3]. We crawled each webpage corresponding to URLs from the WebUI [20] dataset. We skipped URLs without actual contents. During crawling, we filtered out pages with a 404 status by simply determining if the number of page nodes was less than 5 and all page elements were concentrated at the top of the screen. Since our approach requires repeated operations on page nodes, pages need to be re-rendered repeatedly, and the overhead for capturing screenshots is also high. So we performed preliminary filtering of nodes. We filtered elements based on CSS properties to reduce the workload of subsequent steps. Our filtering criteria are as follows. 1) If CSS properties such as display, visibility, and opacity are set to be invisible, we consider the element invisible. 2) If the element's getBoundingClientRect is outside the visible area, we consider it invisible. Based on our observations, the bounding box obtained by getBoundingClientRect is generally larger rather than smaller, making it reliable for preliminary judgment of element visibility. 3) If elements have background or innerText, we preliminarily consider them visible. After this filtering, the number of nodes to be processed is significantly reduced, thereby improving the efficiency of the entire implementation.

The page operation component manipulates the page using JavaScript to achieve DOM transformations and metadata retrieval required at each step of the approach. This component is implemented using the evaluate function provided by playwright to inject JavaScript code. According to HTML5 specifications and browser rendering principles, we inject desired styles by modifying the inline style of elements and adding the !important attribute. We maintain a mapping table to record the original inline style of elements, allowing us to restore the original style when repeated transformations are needed. For hiding and showing elements, we use the visibility property. For pseudo-elements, we control their visibility using opacity and treat them as graphics. To change the GUI background color, we set the background color on the HTML root tag. For text color changes, we set the color property.

The annotation export component is implemented based on Pillow [2] and OpenCV [1]. We use the paste function of Pillow for layer composition and obtain the mask through the absdiff provided by OpenCV combined with matrix operations. Then, we obtain the contour using the findContours function.

Finally, we construct the PixelWeb dataset containing 100,000 samples. Each data sample includes a screenshot of a webpage GUI, along with transparent background images of each GUI element, XYZ axis positions, masks, contours, BBox, and corresponding code and computed style information. Figure 7-9 show examples of our data samples and examples of annotating the same samples using the WebUI approach.

5 Evaluation

In this section, we first assess the quality of the PixelWeb generated by our proposed method through a user study (Section 5.2). We then demonstrate the effectiveness of PixelWeb on various GUI downstream tasks in Section 5.3.

Table 3: Results of User Study

	User1	User2	User3	Total
PixelWeb is much better	85	92	61	238
PixelWeb is better	102	58	99	259
Not sure	53	100	88	241
Baseline is better	39	24	39	102
Baseline is much better	21	26	13	60

5.1 Setup

5.1.1 Models. We use the YOLOv12 [16, 17] series models released in 2025 for experiments on downstream tasks. We try three versions with different parameter scales: 2.6M, 9.3M, and 20.2M, to explore how the dataset performs on different model scales. When training different models and datasets, we set the batch size uniformly to 16, and all other parameters use the default configuration of YOLOv12. In terms of metrics, we use precision, recall, and mAP metrics, distinguishing between mAP@50 and mAP@95.

5.1.2 Baseline. To ensure the fairness of the experiment, while constructing the PixelWeb dataset, we use the annotation method of WebUI to annotate the same data samples, including both BBox and classification annotations, as the baseline for the experiment. The only difference between the PixelWeb and baseline datasets is the method of BBox annotation, allowing for a fair comparison to determine if our annotation method effectively improves BBox quality. We construct PixelWeb and baseline datasets containing a total of 10,000 samples, with 9,000 samples in the training set and 1,000 samples in the validation set. To get an totally precise annotation, we additionally annotate 200 samples to create a test set.

5.2 Annotation Quality

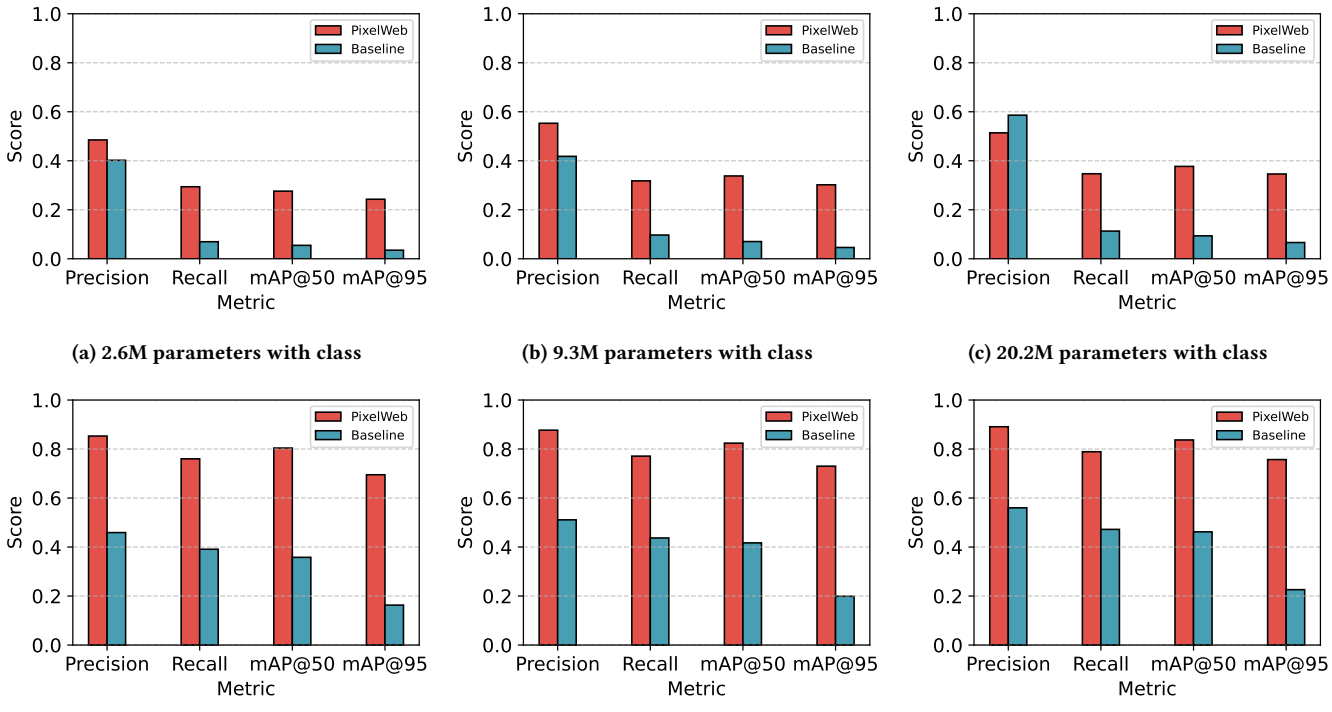
We created 300 slides, each featuring two GUI images with overlaid BBoxes, arranged side by side. The images are randomly selected from the P-web and Baseline datasets, with their order randomized. We engaged three participants with software engineering backgrounds, explained the types of BBox annotation errors, and clarified the evaluation criteria. The participants were then asked to annotate the slides, with options for each: the left sample is slightly better, the left sample is much better, about the same, the right sample is slightly better, or the right sample is much better.

Table 3 shows the results of the user study. PixelWeb received a significantly higher number of "much better" ratings, accounting for 26% of all samples. Overall, 53% of the samples were rated better than the baseline, while only 18% were rated worse. Thus, PixelWeb is subjectively considered superior to the existing baseline.

5.3 Results on Downstream Tasks

Figure 10 display the training results for parameter sizes of 2.6M, 9.3M, and 20.2M, respectively. We observe that across various parameter scales and classification labels, PixelWeb consistently outperforms the Baseline in metrics. As model size increases, advantage of PixelWeb remains significant and stabilizes.

Although under the 20.2M model parameters, the Precision metric of PixelWeb is slightly weaker than that of the Baseline when



(a) 2.6M parameters with class

(b) 9.3M parameters with class

(c) 20.2M parameters with class

(d) 2.6M parameters w/o class

(e) 9.3M parameters w/o class

(f) 20.2M parameters w/o class

Figure 10: Evaluation results for different model sizes with and without class labels

trained with classification labels, the Recall and mAP metrics are significantly better than those of the Baseline. Therefore, overall, PixelWeb is still significantly better than the Baseline. Moreover, as the scale of model parameters increases exponentially, the improvement of the Baseline is not obvious. Even when training the Baseline with the model having the largest parameters in the experiment, all metrics are significantly lower than those of PixelWeb trained with only 1/10 of the parameters. Since the only difference between PixelWeb and the Baseline is the accuracy of BBox annotations, this indicates that the higher quality BBox annotations of PixelWeb indeed lead to a significant improvement in the training effect of the object detection task. Given that the primary difference between PixelWeb and Baseline is the accuracy of BBox annotations, this suggests that the higher quality of annotations in PixelWeb significantly enhances training performance for object detection tasks.

6 Potential for GUI Tasks

PixelWeb shows potential in a wide range of GUI downstream tasks, as shown in Table 1. Accurate BBox annotation is just one of the many labels we provide. Compared to existing datasets, we offer richer annotation information at the element level, including element image and element mask. These new types of annotation information not only enhance various existing GUI downstream tasks but also enable various new GUI downstream tasks. Table 4 demonstrates the new GUI downstream tasks that these novel labels enable. In detection and generation tasks, due to our provision of higher-dimensional information annotations, more fine-grained tasks can be achieved, such as element-level generation tasks and semantic segmentation tasks.

Table 4: Expanding GUI downstream tasks with PixelWeb new annotations

PixelWeb New Annotation	New Tasks
Element Image	GUI element retrieval
	GUI element generation
	GUI layout generation by images
	GUI composting dataset
Element Layer	GUI layout with Z-axis direction
	Advertisement element analysis
Element Mask	GUI instance segmentation
	GUI semantic segmentation
	GUI element shape analysis
	GUI element color generation
Element Contour	GUI instance segmentation
	GUI semantic segmentation
	GUI element shape analysis
Element Computed Style	GUI element generation
	GUI element image2code
	GUI element code2image

7 Conclusion

In this paper, we proposed a novel automated annotation method resulting in the creation of the PixelWeb dataset. This dataset includes BGRA four-channel bitmap annotations and layer position annotations for GUI elements, along with precise bounding box, contour, and mask annotations. Our experimental results demonstrate that PixelWeb significantly outperforms existing GUI datasets, leading to enhanced performance in downstream tasks such as GUI element detection. Looking forward, this dataset can enhance performance in a wider range of tasks, such as intelligent agent interactions and automated web page generation. Furthermore, exploring the potential of transferring these techniques to mobile apps and other non-web GUIs could provide even greater versatility and impact.

References

[1] Opencv. https://opencv.org/.

[2] Pillow. https://pypi.org/project/pillow/.

[3] Playwright. https://playwright.dev/.

[4] Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. Guigle: A gui search engine for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 71–74. IEEE, 2019.

[5] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. Vins: Visual search for mobile user interface design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2021.

[6] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[7] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[8] Peitong Duan, Chin-Yi Cheng, Gang Li, Bjoern Hartmann, and Yang Li. Uicrit: Enhancing automated design evaluation with a ui critique dataset. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–17, 2024.

[9] Sidong Feng, Suyu Ma, Han Wang, David Kong, and Chunyang Chen. Mud: Towards a large-scale and noise-filtered ui dataset for modern style ui modeling. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2024.

[10] Kamal Gupta, Justin Lazarow, Alessandro Achille, Larry S Davis, Vijay Mahadevan, and Abhinav Shrivastava. Layouttransformer: Layout generation and completion with self-attention. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1004–1014, 2021.

[11] Naoto Inoue, Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. Layoutdm: Discrete diffusion model for controllable layout generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10167–10176, 2023.

[12] Xiang Kong, Lu Jiang, Huiwen Chang, Han Zhang, Yuan Hao, Haifeng Gong, and Irfan Essa. BLT: Bidirectional Layout Transformer for Controllable Layout Generation. http://arxiv.org/abs/2112.05112, 2022. Retrieved October 14, 2024.

[13] Gang Li, Gilles Baechler, Manuel Tragut, and Yang Li. Learning to denoise raw mobile ui layouts for improving datasets at scale. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2022.

[14] Zhangheng Li, Keen You, Haotian Zhang, Di Feng, Harsh Agrawal, Xiujun Li, Mohana Prasad Sathya Moorthy, Jeff Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui 2: Mastering universal user interface understanding across platforms. *arXiv preprint arXiv:2410.18967*, 2024.

[15] Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*, 2024.

[16] Yunjie Tian, Qixiang Ye, and David Doermann. Yolov12: Attention-centric real-time object detectors. *arXiv preprint arXiv:2502.12524*, 2025.

[17] Yunjie Tian, Qixiang Ye, and David Doermann. Yolov12: Attention-centric real-time object detectors, 2025.

[18] Jianqiang Wan, Sibo Song, Wenwen Yu, Yuliang Liu, Wenqing Cheng, Fei Huang, Xiang Bai, Cong Yao, and Zhibo Yang. Omniparser: A unified framework for text spotting key information extraction and table recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15641–15653, 2024.

[19] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R Lyu. Automatically generating ui code from screenshot: A divide-and-conquer-based approach. *arXiv preprint arXiv:2406.16386*, 2024.

[20] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P. Bigham. Webui: A dataset for enhancing visual ui understanding with web semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2023.

[21] Shuhong Xiao, Yunnong Chen, Jiazhi Li, Liuqing Chen, Lingyun Sun, and Tingting Zhou. Prototype2code: End-to-end front-end code generation from ui design prototypes. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 88353, page V02BT02A038. American Society of Mechanical Engineers, 2024.

[22] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. Uied: a hybrid tool for gui element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1655–1659, 2020.

[23] Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer, 2024.