CSCD01: Deliverable 2

Team 32: CCF

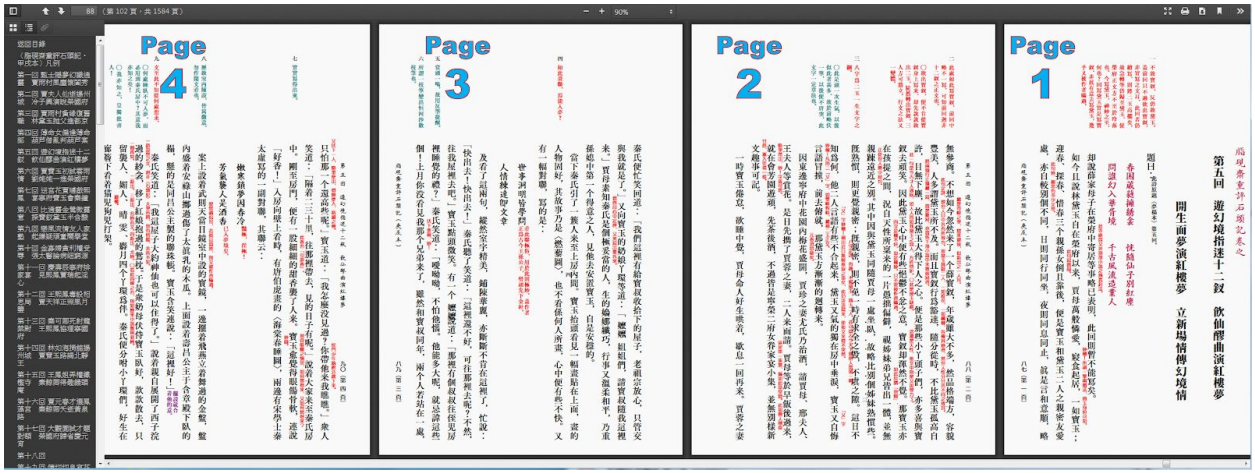
Angela Zhu
Dasha Besshaposhnikova
Michelle Pasquill
Mohammed Osumah

Table of Contents

ISSUE #1 - Horizontal Scrolling

Link to issue: https://github.com/mozilla/pdf.js/issues/11154

This issue is a new function suggestion for horizontal scrolling. Currently there is an option to enable horizontal scrolling mode in PDF.js, but the direction of the pages goes from left to right. The user who raised the issue was suggesting a right to left scrolling mode for documents that are intended to be read from right to left, such as Chinese/Japanese/Korean vertical texts.



How to reproduce the issue:

FireFox (Can run in any other browser using pdf.js as an extension)

1. Open any pdf with multiple pages
2. Enable horizontal scrolling by clicking on the » button in the top right corner and selecting Horizontal Scrolling

Source Code

1. Clone the source code from link https://github.com/CSCD01-team32/pdf.js/
2. Install Node.js and install the gulp package (more information in README.md)
3. Use gulp server and open http://localhost:8888
4. Open any pdf with multiple pages
5. Enable horizontal scrolling by clicking on the » button in the top right corner and selecting Horizontal Scrolling

Estimate of amount of work to complete:

We estimated that this feature would take 2-3 days of work and at least 2 developers working together since the feature would span multiple files and things like mouse/keyboard events, thumbnail ordering, and UI changes would need to take place. We thought this would be a good issue for our team to work on because it would mainly involve examining the code and implementing the reverse of the existing scrolling feature, which is something our team would be comfortable with. However, we did not select to implement this feature just because of the amount of time it would take and the amount of files that we would need to change.

Some of the files involved in this feature are:
pdf_presentation_mode.js - Deals with page switches according to different events.
pdf_thumbnail_viewer.js - renders the thumbnail view
ui_utils.js - Deals with scrolling and scroll mode

ISSUE #2 - WAI-ARIA

Link to issue: https://github.com/mozilla/pdf.js/issues/9556

This function was suggested by a user who utilizes screen readers. Pdf.js is not completely up to accessibility standards, and as such they have suggested additional support the viewer needs. Currently the viewer allows one to tab through its toolbar. It was suggested that keyboard shortcuts and arrow key functionality also be added to toolbar navigation. For example, adding a keyboard shortcut for toggling the sidebar navigation.

Estimate of amount of work to complete:

We estimated that this feature would take [insert amount of time]. This is because the feature would require us to modify keyboard events in several files. We looked at this issue because accessibility features are important features and many of them are not very workload heavy. Implementing this accessibility feature requires modifying and adding on to existing features and thus manageable in our time frame. The reason we did not select this issue is because of how long it would take us to modify all the key events. Keyboard shortcuts would require us to modify existing keyboard functionality; the viewer allows for navigating the pdf with arrow keys, and thus would need to switch focus when using a screen reader on the toolbar. Keyboard shortcuts would also require us to work around existing shortcuts and create through documentation.

ISSUE #3 - Converting Colors in ‘web/viewer.css’ to CSS Variables

Link to issue: https://github.com/mozilla/pdf.js/issues/11572

This issue was brought up by one of the top contributors to PDF.js. He was suggesting that the css file web/viewer.css should be altered so that all the colours used are represented by CSS variables, to make it easier for reskinning the viewer.

Currently the css colours are defined like this:

```
#viewerContainer.pdfPresentationMode:-ms-fullscreen {
  top: 0px !important;
  overflow: hidden !important;
}

#viewerContainer.pdfPresentationMode:-ms-fullscreen::-ms-backdrop {
  background-color: rgba(0, 0, 0, 1);
}

#viewerContainer.pdfPresentationMode:fullscreen {
  top: 0px;
  border-top: 2px solid rgba(0, 0, 0, 0);
  background-color: rgba(0, 0, 0, 1);
  width: 100%;
  height: 100%;
  overflow: hidden;
  cursor: none;
  user-select: none;
}
```

Using rbga() to set the colour value, and this method is used over 200 times in the file, even though the amount of colours used is limited, with a lot of colours being reused.

Estimate of amount of work to complete:

We estimated that the work required for this fix is maximum 1 hour, since CSS variables are not very complicated and only involves declaring the rgba values used as variables in :root. This is one of the issues we chose to work on, and further justification can be found in the *Rationale for selected issues* section of this deliverable.

ISSUE #4 - Tab-order

Link to issue: https://github.com/mozilla/pdf.js/issues/9557

This is an accessibility issue with the UI viewer in pdf.js. The issue covers two different problems, there are tabindex values greater than 0, and there are some tabbable elements that aren't encapsulated in an ARIA landmark.

First, the tabindex values allow for keyboard navigation for anyone who requires the use of a screen reader or who has mobility problems. Currently the tabindex values are manually set as different positive values ranging from 0 to around 100. The increasing values dictate the order that the elements should be tabbed through. This is bad practice in general, since inserting a single element into the tab order may require that all tab indexes for later elements be manually increased. For accessibility reasons this is bad since it can cause an unexpected tab order which can cause it to appear as if the items are skipped entirely.

Another problem brought up in this issue, is that there are no ARIA landmark roles in the viewer. ARIA landmark roles provide the screen reader user with the ability to skip to different main sections in the document, rather than having to tab through everything to get to the desired section. Currently there are no landmarks, meaning the user needs to tab through the entire navigation bar before they can actually read the pdf.

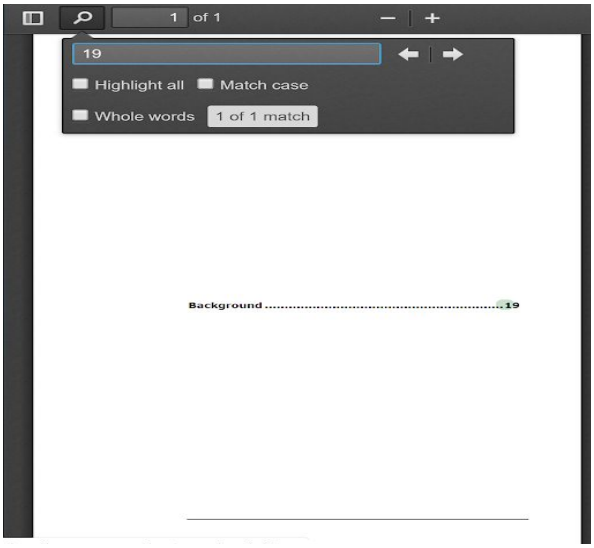
Estimate of amount of work to complete:

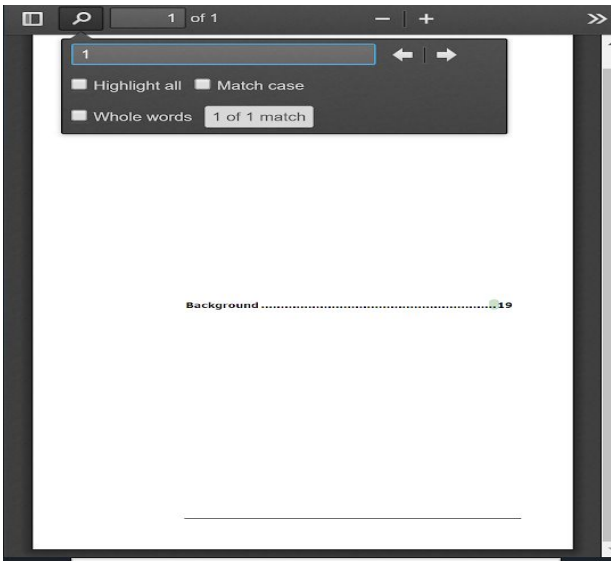
The work for this issue would not be too much. Probably no longer than 30 minutes to find screen reader software and learn to use it for testing purposes. Then about 1-2 hours to change tabindex values, add ARIA landmarks, and to change the javascript to ensure that drop-down type menus have focus moved to them on opening.

ISSUE #5 - CTRL-F Highlighting Problem

Link to issue: https://github.com/mozilla/pdf.js/issues/9752

There is an issue with the highlighting of text when searching for text in the pdf with the CTRL-F option. You can see in the screenshot of the pdf file below (bottom left) that we opened with pdf.js, when searching for the string “19” with CTRL-F, the highlighting is off. The string “19” in the pdf file should be highlighted, but instead you see that the string “.. 1” is highlighted instead. The highlighting does not match the text in this case. When searching for the string “1” it also wasn’t highlighted correctly. This is evident in the screenshot below (bottom right).





How to reproduce the bug:

FireFox (Can run in any other browser using pdf.js as an extension)

1. Download the file issue9752.pdf in our github folder for deliverable 2
2. Open the file in a firefox browser
3. CTRF-F and search for strings “19” or “1”

Source Code

1. Download the file issue9752.pdf in our github folder for deliverable 2
2. Clone the source code from link https://github.com/CSCD01-team32/pdf.js/
3. Install Node.js and install the gulp package (more information in README.md)
4. Use gulp server and open http://localhost:8888
5. Open the downloaded file
6. CTRL-F and search for strings “19” or “1”

Estimate of amount of work to complete:

Initially we assumed it was a functional issue, where the highlighting was a little off. We initially thought it would only take approximately 2-3 hours worth of work to diagnose this issue, but as we continued to dive deeper into finding the areas of the code where the bug might be, we realized it was a bigger issue involving the rendering of the text layer, which could take days to fix.

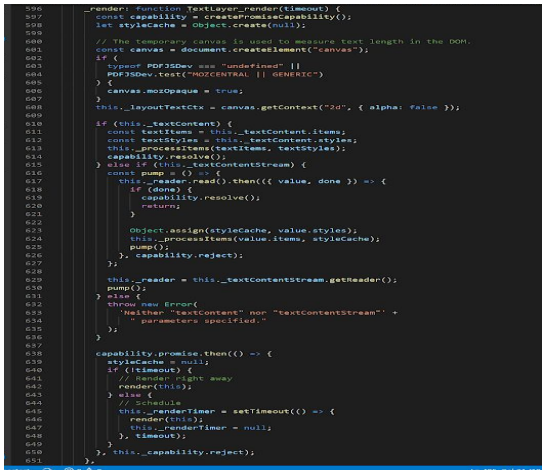
When testing the code we made the text layer visible in order to see whether the highlighting was off on it. We discovered that the highlighting is correct in the text layer, so we're pretty sure the problem has to do with the text layer not being rendered correctly. In the screenshots below, you can see the pdf when the text layer is visible and when it is not. The light gray in the pdf in the bottom right screenshot is the text layer.

We suspect the issue has to be in the file text_layer_builder.js in the src/web folder, which imports renderTextLayer from text_layer.js in the src/display folder. In the images below, we can see the function renderTextLayer and we can see it just initializes a TextLayerRenderTask object and runs the _render method. In the image below to the right you can see the code for _render.

```
/**
 * Starts rendering of the text layer.
 *
 * @param {TextLayerRenderParameters} renderParameters
 * @returns {TextLayerRenderTask}
 */
function renderTextLayer(renderParameters) {
  var task = new TextLayerRenderTask({
    textContent: renderParameters.textContent,
    textContentStream: renderParameters.textContentStream,
    container: renderParameters.container,
    viewport: renderParameters.viewport,
    textDivs: renderParameters.textDivs,
    textContentItemsStr: renderParameters.textContentItemsStr,
    enhanceTextSelection: renderParameters.enhanceTextSelection,
  });
  task._render(renderParameters.timeout);
  return task;
}

return renderTextLayer;
})();

export { renderTextLayer };
```



```
_render: function TextLayer_render(timeout) {
  const capability = createPromiseCapability();
  let styleCache = Object.create(null);

  // The temporary canvas is used to measure text length in the DOM.
  const canvas = document.createElement("canvas");
  if (
    typeof PDFJSDev === "undefined" ||
    PDFJSDev.test("MOZCENTRAL || GENERIC")
  ) {
    canvas.mozOpaque = true;
  }
  this._layoutTextCtx = canvas.getContext("2d", { alpha: false });

  if (this._textContent) {
    const textItems = this._textContent.items;
    const textStyles = this._textContent.styles;
    this._processItems(textItems, textStyles);
    capability.resolve();
  } else if (this._textContentStream) {
    const pump = () => {
      this._reader.read().then(({ value, done }) => {
        if (done) {
          capability.resolve();
          return;
        }

        Object.assign(styleCache, value.styles);
        this._processItems(value.items, styleCache);
        pump();
      }, capability.reject);
    };

    this._reader = this._textContentStream.getReader();
    pump();
  } else {
    throw new Error(
      'Neither "textContent" nor "textContentStream"' +
        " parameters specified."
    );
  }

  capability.promise.then(() => {
    styleCache = null;
    if (!timeout) {
      // Render right away
      render(this);
    } else {
      // Schedule
      this._renderTimer = setTimeout(() => {
        render(this);
        this._renderTimer = null;
      }, timeout);
    }
  }, this._capability.reject);
},
```

Rationale for selected issues

(Submit an argument for why these are the best items for you to work on. This includes estimating the effort required to implement each change, and identifying any anticipated risks.)

Chosen Issue: Converting Colors in ‘web/viewer.css’ to CSS Variables

See Issue #3 for more information

We ended up choosing this issue instead of the Ctrl+F highlighting problem when we discovered that the cause of the highlighting problem was more complicated than we initially thought. This CSS variables issue, on the other hand, is very straightforward and allowed us to make a useful fix for PDF.js. With CSS variables, the code becomes much cleaner and easier to build upon for future developers. It also allowed us to fix a second issue on time for this deliverable.

The anticipated risk for this issue is that a developer had left a comment on the issue expressing concern that the CSS variables might add overhead to the loading of the pdf viewer. Upon doing some research we discovered that many sources state that CSS variables function well, and are much better than reusing the same values many times throughout CSS files. Since we won’t be adding a large amount of variables anyway, we decided it wouldn’t make much of a difference to the overhead.

Acceptance test

Since there is no way of coding an acceptance test for css variables, it suffices to open web/viewer.js and in the :root pseudo class with the variables under the color comment, search each of the variables in the file. If you see the color variables being reused, and the pdf viewer looks unchanged from the previous version, then the implementation has been successfully completed.

Chosen Issue: Tab-order

See Issue #4 for more information

We chose to work on this issue since it would not take too long to fix and some of our team members have had experience with making accessible web content before. We also believe that accessibility is an important issue, and this problem could make PDF.js frustratingly unusable for people who need to use screen readers. Additionally, it is bad practice to put positive tab indexes, so by fixing this it makes the viewer easier to

change in the future.

Anticipated risks of fixing this issue are that we needed to change all tabindex values to 0, meaning the tab order may not follow expectations if the DOM elements aren't organized well. Additionally, some javascript may modify tabindex values directly, and changing them could cause other problems. These risks can be mitigated by reordering DOM elements to follow a logical order, as well as checking the javascript for any potential tabindex or ARIA landmark manipulations.

Acceptance testing

From our research, there does not seem to be any way of automating testing for ensuring a logical tab order or checking ARIA landmarks. Instead, we manually used a screen reading plugin for Google Chrome called "ChromeVox". This plug in allowed us to check that the tab ordering makes sense and also to test that the ARIA landmarks are working. Using this to test, we were able to find any issues where the DOM does not flow in a logical left to right order and modify it to fix this. This also allowed us to find an additional problem with the fact that opening a sub-menu sometime does not move focus to that sub-menu. This means that someone with a screen reader may click "Enter" to open a menu, expecting the next tabbable element to be from the sub-menu when it is not, leading to confusion.

Additionally, we used another plug-in called "axe" which is a tool with automated accessibility testing which can be used in the in-browser developer tools. This tool checks any violations of web accessibility guidelines and tells you how many there are and what element in the DOM violates it. The tool provided feedback regarding the positive tabindex values and lack of ARIA landmarks. Screenshots of the tool being used can be found on GitHub under the deliverable 2 acceptance test folder.

Technical Commentary

(Write a brief technical commentary on how your changes affect the design and/or the code of the project. List ALL relevant source code files that were added, modified, or removed as part of your implementation work.)

Chosen Issue: Converting Colors in ‘web/viewer.css’ to CSS Variables

See Issue #3 for more information

For this issue, the only file we needed to update was web/viewer.css. We converted the numerous repeated rgba() to define colours as variables. These changes make the design of the code for this file in the project more cleaner and easier for reskinning.

It shouldn’t affect the functionality of the project because everything should still work as it is, seeing as we just converted the repeated colours to variables, but didn’t change the colours for the classes and IDs.

Chosen Issue: Tab-order

See Issue #4 for more information

For this issue, we had to modify the HTML file for the viewer as well as the javascript.

All tabbable elements (with the exception of the page thumbnail viewer which uses aria labels) were managed using tabindex. However, with the amount of elements, the values were going to incredible high numbers. Notably, elements in the find bar had values in the 90s while the find button for toggling the bar had a value under 10. This caused inconsistency with tabbing order. For example, the elements in the sidebar menu had high numbers, and when opened, the tab would continue through the toolbar instead of going into the newly opened sidebar menu.

The HTML file modified was web/viewer.html. We needed to change the ‘tabindex’ values to follow WAI-ARIA guidelines. There were 43 instances where ‘tabindex’ was set and needed to be changed to “tabindex=0”. Making them all “tabindex=0” brings them all into the tab order on the page without specifically specifying an order, the elements now just follow the order which they appear in the DOM. An additional change needed to be made for the organization of the DOM. Their toolbar has three sections but they were not represented in order in the DOM, so they needed to be reorganized so the tab order makes sense.

Additionally, ARIA landmark roles were added to two elements in the HTML document. The ‘toolbar’, which is the top bar in the viewer, was given ‘role=”navigation”’. The ‘viewerContainer’, which contains the display for the PDF page, was given ‘role=”main”’. This allows the screen reader user to quickly skip to the navigation bar or the main document.

While some javascript files (such as pdf_thumbnail_viewer.js) did modify ARIA labels in the html file, ARIA landmarks were not included and tabindex attributes were only handled in the HTML. Due to this, no other part of the JS was affected by the HTML changes.