Text selection and highlighting in a Pas2js PDF viewer

Michaël Van Canneyt

December 10, 2023

Abstract

In previous articles we introduced a way to show a PDF in the browser, and to search in a PDF. In this article we add a missing feature: highlighting search results in the text and text selection.

1 Introduction

In a series of articles on PasJS we demonstrated how to show a PDF in the browser, and how to search for a text in the shown PDF.

This can be done relatively easy using PDF.js, the Javascript PDF displaying library by Mozilla. Later on we added the capability to search in a series of PDFs using a server-side mechanism and an indexer written in Free Pascal. The result is the basis for the Blaise Pascal Magazine library, a searchable library of articles published in Blaise pascal magazine: a program that works both offline and online.

These demo programs had 2 drawbacks: The first drawback was that the search mechanism was limited to finding the text and displaying the correct page in the PDF. The second drawback was that it was impossible to select (and possibly copy) text in the PDF.

Luckily, the PDF.js API contains some calls that solve both these problems. In this article we show how to use one of these calls, thereby solving both problems at once.

2 Ugrading PDF.js

Before diving in, a small detour is needed: In between the publication of the various programs that show the workings of the PDF.js library, the library has changed in a way that necessitates changes in the Pascal code and the HTML. These changes are not extensive, but are necessary or your program will stop working - if it has not already stopped working, which is likely if you used the online distribution of PDF.js through some CDN.

Previously, the PDF.js library exposed a global variable `pdfjsLib` which was defined in the programs as:

```
var
  pdfjsLib : TPDFJSStatic; external name 'pdfjsLib';
```

The PDF.js library is now created as a Javascript module, similar to a dynamically loadable library. The library filename extension was also changed to `.mjs` to reflect this. As a result, the above variable is no longer defined if you load it as a regular script, because the browser will refuse to load the script.

The solution is to load the library as a module: This simply means that you must add a `type` attribute with the value `module` to the script tag that includes PDF.js, for example as follows:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/pdf.js/4.0.269/pdf.mjs"
        type="module">
</script>
```

Then the script will be correctly loaded and the global variable will be defined.

If you are using a private copy of the legacy build of PDF.js, then there is no need to change anything.

3 A selection and highlighting mechanism

The reason why the first versions of the PDF viewer did not allow highlighting and selection is because basically, the PDF is rendered as a bitmap on a HTML canvas, including the text. It should be clear that the drawn text bitmap cannot be selected (unless using OCR), and that some other mechanism is needed.

Luckily, PDF.js offers a solution: it contains a mechanism to render the text elements of a PDF file as HTML, overlaid on the canvas. This HTML is styled so it is completely transparent, and the user does not see it. But it is actual HTML that is part of the HTML page's DOM, and can be manipulated with the usual DOM and CSS techniques. Additionally, when selecting something in the browser, the 'hidden' html is taken into account: the 'selected' pseudo-element will also work, and can be styled: by changing the background color of the 'selected' pseudo-element, the selected text can be made visible.

This means that if we use the PDF.js API for rendering text, we have our text selection mechanism.

Since the rendered HTML contains the actual text (but simply rendered invisibly), it means that when we searched for a text and a page containing a match of the text is displayed, we can traverse the created HTML and highlight matches in the text.

In the below, we'll show how to do this.

4 Allowing selection: Rendering page text as HTML.

The PDF.js API has 2 calls that create or update the HTML DOM with the text of the PDF. Both are implemented using a background task, and they both return an instance of this background task:

```
function renderTextLayer(params : TPDFJSRenderTextLayerParameters) : TPDFTextLayerRenderTask;
function updateTextLayer(params : TPDFJSUpdateTextLayerParameters) : TPDFTextLayerRenderTask;
```

The call that interests us is the `RenderTextLayer` task. It accepts the following parameter object:

```
TPDFJSRenderTextLayerParameters = class
  textContentSourceStream : TJSReadableStream; external name 'textContentSource';
  textContentSourceItems: TTextContent; external name 'textContentSource';
  container: TJSHTMLElement;
  viewport : TPDFPageViewport;
  isOffscreenCanvasSupported : Boolean;
  textDivs : TJSHTMLElementArray;
  textDivProperties : TJSMap;
  textContentItemsStr : TStringDynArray;
end;
```

The first 5 fields in this class are input:

textContentSourceStream The text contents of the PDF as a stream.

textContentSourceItems The text contents of the PDF as returned by the `getTextContent` call of the TPDFPageProxy object.

container The HTML element in which to render the text.

viewport The viewport using which the PDF was rendered.

isOffscreenCanvasSupported Set to `True` if the PDF layer can use an offscreen canvas. (needed to determine text widths)

The last 3 fields of the class are output: they will be filled after the `renderTextLayer` call did its work. Initially they should be set to empty arrays.

textDivs An array with the generated HTML elements;

textDivProperties an array with the properties needed to generate the HTML elements.

textContentItemsStr The raw texts of the generated HTML elements.

We'll demonstrate this call in the PDF search example we presented earlier. In that example, the user could enter an URL or select a PDF file which would then be shown, and which could be searched. As a reminder, figure 1 on page 4 shows what the application looked like.

We'll reuse and expand that example, the source code will of course again be available for you.

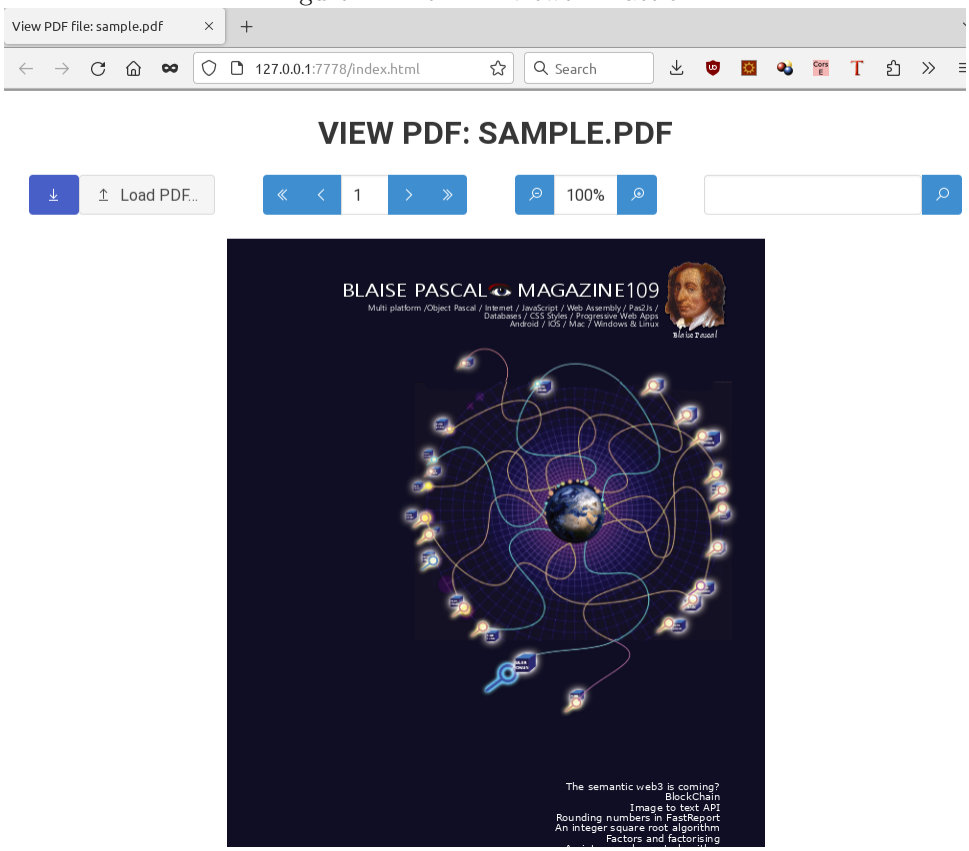
In order to use the `renderTextLaye` call, we need to add additional elements to the HTML for our PDF viewer. Where previously we had the following HTML in which the PDF was shown:

```
<div class="is-flex is-justify-content-center">
   <canvas id="PDFCanvas" height="737" width="538"></canvas>
</div>
```

We now need an extra element that will be used to overlay the text (with ID `pdfTextLayer`), and another one to carry an extra style parameter (with id `pdfViewer`):

```
<div class="is-flex is-justify-content-center">
  <div id="pdfViewer" style="position: relative">
    <div class="canvaswrapper" >
```

Figure 1: The PDF viewer in action



```
      <canvas id="PDFCanvas" height="737" width="538"></canvas>
    <div>
    <div id="pdfTextlayer" class="textLayer">
    </div>
  </div>
</div>
```

The 2 IDs will be bound to 2 variables in our program:

```
TMyApplication = class(TBrowserApplication)
  divpdfViewer,
  FTextlayer,
  lblFileLocation,
  lblZoom : TJSHTMLElement;
  // ...
end;
```

The rendering of a page of the PDF document was done in the `RenderPage` method of our program: in this method, the `GetPage` call is used to retrieve a proxy object for a PDF page, and the `render` method is used to actuall render the page in the canvas:

```
procedure TMyApplication.renderPage(aNum: Integer);

function renderOK(aValue : JSValue) : JSValue;

Var
  N : Integer;

begin
  FPageRendering:=false;
  if (FPageNumPending <> -1) then
    begin
    N:=FPageNumPending;
    FpageNumPending:=-1;
    renderPage(N);
    end;
  Result:=True;
end;

function havePage (aValue : JSValue) : JSValue;

var
  page : TPDFPageProxy absolute aValue;
  viewport : TPDFPageViewport;
  renderContext: TPDFRenderParams;
  renderTask : TPDFRenderTask;
  viewportParams : TViewportParameters;

begin
  viewportParams:=TViewportParameters.new;
  viewportParams.scale:=FScale;
  viewport:=page.getViewport(viewportParams);
  Fcanvas.height := viewport.height;
```

```
  Fcanvas.width := viewport.width;
  renderContext:=TPDFRenderParams.New;
  renderContext.canvasContext:=Fctx;
  renderContext.viewport:=viewport;
  renderTask:=page.render(renderContext);
  renderTask.promise.&then(@renderOK);
  Result:=True;
end;

begin
  FpageRendering:=True;
  pdfDoc.getPage(aNum).&then(@HavePage);
  edtPageNo.Value:=IntToStr(anum);
end;
```

As you can see in the code of the `havePage` routine, the parameters for the PDF page render task are the same as the ones for the `RenderTextLayer` call. It makes therefore sense to put them in a record which is declared in our application class:

```
TCurrentPageInfo = record
  page : TPDFPageProxy;
  viewport : TPDFPageViewport;
  renderContext: TPDFRenderParams;
  renderTask : TPDFRenderTask;
  viewportParams : TViewportParameters;
  end;

TMyApplication = class(TBrowserApplication)
  divpdfViewer,
  FTextlayer,
  lblFileLocation,
  lblZoom : TJSHTMLElement;
  FCurrentPageInfo : TCurrentPageInfo;
  // ...
end;
```

We now can rewrite the `havePage` procedure so it uses the newly introduces record to store the information needed to render the page:

```
function havePage (aValue : JSValue) : JSValue;

var
  page : TPDFPageProxy absolute aValue;

begin
  FCurrentPageInfo.Page:=page;
  FCurrentPageInfo.viewportParams:=TViewportParameters.new;
  FCurrentPageInfo.viewportParams.scale:=FScale;
  FCurrentPageInfo.viewport:=page.getViewport(FCurrentPageInfo.viewportParams);
  Fcanvas.height := FCurrentPageInfo.viewport.height;
  Fcanvas.width := FCurrentPageInfo.viewport.width;
  FCurrentPageInfo.renderContext:=TPDFRenderParams.New;
  FCurrentPageInfo.renderContext.canvasContext:=Fctx;
  FCurrentPageInfo.renderContext.viewport:=FCurrentPageInfo.viewport;
```

```
  FCurrentPageInfo.renderTask:=page.render(FCurrentPageInfo.renderContext);
  FCurrentPageInfo.renderTask.promise.&then(@renderOK);
  Result:=True;
end;
```

When the rendering is done, the `RenderOK` procedure is called, and there we can now add the necessary code to render the text. We start by getting the actual text using the `getTextContent` call. The observing reader will note that the `getTextContent` call is the same call that was used to retrieve the PDF text to search in the PDF.

The renderOK call then becomes:

```
function renderOK(aValue : JSValue) : JSValue;

Var
  N : Integer;

begin
  FPageRendering:=false;
  if (FPageNumPending <> -1) then
    begin
    N:=FPageNumPending;
    FpageNumPending:=-1;
    renderPage(N);
    end;
  Result:=True;
  FCurrentPageInfo.page.getTextContent().&Then(@RenderText);
  divpdfViewer.style.setProperty('--scale-factor',FloatToStr(FScale))
end;
```

The `pdfViewer` element gets a style property that sets a `scale-factor` CSS variable: this variable is needed by the CSS that is generated by PDF.js.

The value of the variable is set to the current scale used to draw the PDF. Forgetting to set it (or setting it to a wrong value) will result in HTML that is not properly positioned over the PDF image.

When the text is retrieved, the `RenderText` callback which we supplied to the Javascript Promise object returned by `getTextContent`, will be called.

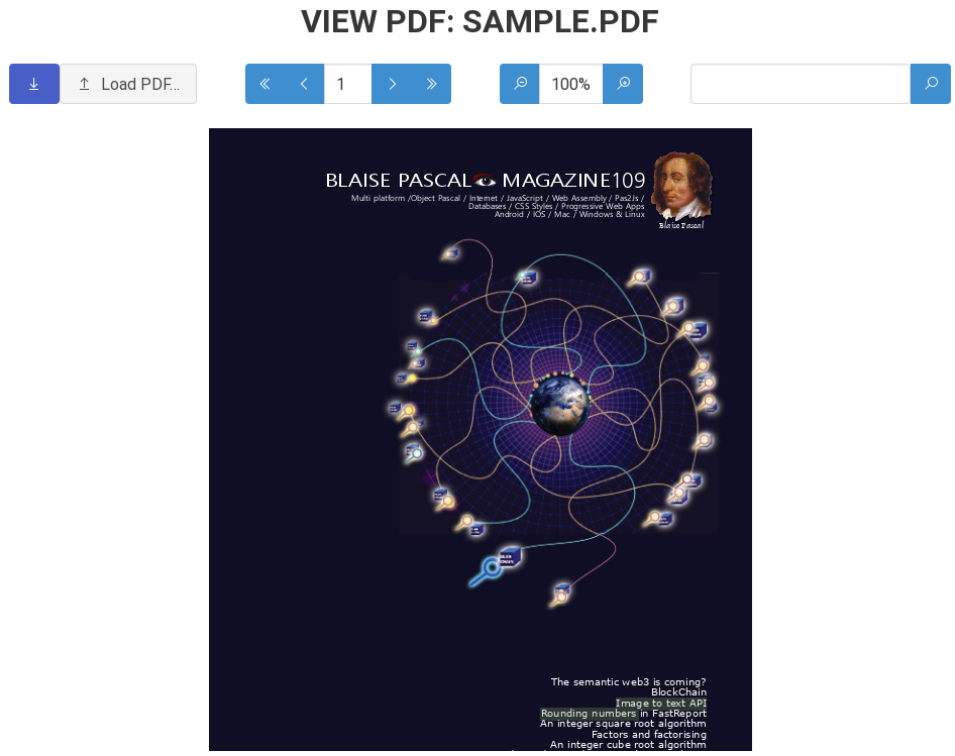
The callback simply prepares the arguments for the `RenderTextLayer` API call, using the `FCurrentPageInfo` and the result of the promise:

```
Function TMyApplication.RenderText(aValue: JSValue) : JSValue;

Var
  aContent : TTextContent absolute aValue;
  aTextRender : TPDFJSRenderTextLayerParameters;

begin
  FTextlayer.InnerHTML:='';
  aTextRender:=TPDFJSRenderTextLayerParameters.New;
  aTextRender.container:=FTextlayer;
  aTextRender.isOffscreenCanvasSupported:=true;
  aTextRender.textContentSourceItems:=aContent;
  aTextRender.viewPort:=FCurrentPageInfo.viewport;
  pdfjsLib.renderTextLayer(aTextRender).promise.&then(@TextDone);
```

Figure 2: Selection in the PDF viewer in action




```
end;
```

With the above code in place, the capability to select and copy text is almost there: the necessary HTML is generated and positioned on the correct place in the HTML page. What is missing is some CSS that is required by the generated HTML. The interested reader can find it in the `viewer.css` file, which we include in the HTML page with a simple `link` HTML element:

```
<link rel="stylesheet" href="viewer.css">
```

The result of all this can be seen in figure 2 on page 8: some text (over multiple lines) is selected in the overview of articles. If the scale of the PDF is changed, then the PDF is re-rendered, and the text is re rendered as well: Since the scale CSS variable is set before rendering, the 2 will always be in sync.

5 Highlighting search matches

What is still missing is the highlighting of search results once a page has been rendered as a result of a search operation. Adding this functionality is not so difficult: In the `TextDone` callback to the `renderTextLayer` call, we have the opportunity to do so.

Looking at the generated HTML by PDF.js, we can see that it is simply a flat series of `span` HTML elements with some positioning applied: the span elements only contain text. To highlight the text, we can simply loop over the `span` elements and check if the span contains the text. Basically, this is the same loop as done in

the search algorithm, the difference is that now we loop over the generated HTML instead of the structures returned by the `getTextContent` call.

The following code checks whether a search was performed. If not, it exits at once. If a search was performed, it prepares the regular expression used for the search, and loops over all `span` tags, calling `Highlight` for every `span` element:

```
Function TMyApplication.TextDone(aValue: JSValue) : JSValue;

var
  El : TJSELement;
  aRegex : TJSRegExp;
  aReg,aTerm : String;

begin
  aTerm:=edtSearch.Value;
  if aTerm='' then exit;
  aReg:=Format('%s',[aTerm]);
  aRegEx:=TJSRegExp.New(aReg,'gi');
  El:=FTextlayer.firstElementChild;
  While Assigned(El) do
    begin
    if (el is TJSHTMLElement) and SameText(El.tagName,'span') then
      HighLight(TJSHTMLElement(El),aRegex);
    El:=El.nextElementSibling;
    end;
end;
```

In the `highlight` routine, we modify the inner HTML of the `span` element. If a match (or multiple matches) is found, then we surround the match with a new `span` element which we give a 'highlight' CSS class: For example, if the search text is 'integer' then the following `span` element:

```
<span>An integer square root algorithm</span>
```

becomes

```
<span>An <span class="highlight">integer</var> square root algorithm</span>
```

The 'highlight' class is picked up by the CSS to color the text background in a transparant way, and the text in the canvas below will appear as highlighted.

This mechanism can be coded as follows: It is in fact a simple loop using the regular expression: as long as the reguar expression finds a match, the text between the match and the end of the previous match is added to the `span` element as-is, and then the matched text is added embedded in a new span element. If there are no more matches, the routine appends the remaining text.

```
Procedure TMyApplication.HighLight(El : TJSHTMLElement; aRegex : TJSRegexp);

Var
  S,aText,aLeft : String;
  Matches : TStringDynArray;
  aLast : Integer;
  aSpan : TJSHTMLElement;
```

```
begin
  aText:=El.innerText;
  Matches:=aRegex.exec(aText);
  aLast:=1;
  // We exit at once if there is nothing to do.
  if not Assigned(Matches) then
    exit;
  // Clear the HTML
  EL.InnerHTML:='';
  While Assigned(Matches) do
    begin
    // Add preceding text
    S:=Copy(aText,aLast,aRegex.lastIndex-Length(Matches[0]));
    // Create span.
    El.AppendChild(document.createTextNode(S));
    aSpan:=TJSHTMLElement(document.createElement('span'));
    aSpan.InnerText:=Matches[0];
    aSpan.className:='highlight';
    El.AppendChild(aSpan);
    // Search again.
    aLast:=aRegex.LastIndex+1;
    Matches:=aRegex.exec(El.innerText);
    end;
  // Append last text.
  if aLast<length(aText) then
    begin
    S:=Copy(aText,aLast,Length(aText)-aRegex.lastIndex);
    El.AppendChild(document.createTextNode(S));
    end;
end;
```

With this, the highlighting is complete. The result can be seen in figure 3 on page 11.

6 Conclusion

Adding a text layer to the PDF.js viewer allows to implement selection and copy-and-paste operations out of the box. As shown in this article, it can also be used to implement in a rather straightforward manner the highlighting of search terms on a page. The mechanism is not perfect, as the PDF text will sometimes be split into different PDF elements (for exaple when formatting changes): The HTML will consequently be split over HTML tags. For common situations the mechanism is perfectly suitable.

Figure 3: Result highlighting in the PDF viewer in action