在图片上快速编辑公式的编辑器设计

李沪纲

2023 年 *4* 月 *1* 日

摘要

目前市场上的大多数图片编辑器都不支持公式编辑功能，而为广泛使用的 *Microsoft Word* 仅能在文档中添加公式，无法在图片上添加公式。学术界常用的排版系统 LaTeX 也只能生成文档，但同样无法在现有的图片上编辑数学公式，这给用户带来了一定的不便。为了解决这一问题，本论文设计了一个能够快速在图片上编辑公式的编辑器，并提供了相关工程实现。

关键词：公式；公式编辑器；图片处理

目录

1 绪论

1.1 研究背景

如今，有许多工具能帮助人们在电脑上编辑公式。例如，*Microsoft Word* [1] ，$\LaTeX$ [2] 等。但是，这些工具都只能编辑或生成文档，不能很方便的对图片进行操作。到目前为止，在图片上编辑公式最方便的办法就是使用多数图片编辑器中自带的画笔，结合手写板进行手写，但是，手写板毕竟只是字迹，写出来的公式带有个人的笔风，并不像 *Word* 和 $\LaTeX$ 所生成的那么标准，并且容易写错和被人误认。而且，没有手写板时，只用鼠标写出来的字迹不堪入目。

1.2 研究目的与意义

对此，我们意图发明一个能快速在图片上编辑公式的编辑器，能帮助广大教师学生更好的批改作业、分享题目思路解答，提高效率，节省时间。

1.3 创新点

在 *Microsoft Word* 中，编辑公式需要用鼠标去点击，虽说操作起来比较简单，上手难度低，但是降低了工作的效率。而在排版系统 $\LaTeX$ 中，公式是用键盘键入的，就像这样 `x=\frac{-b \pm \sqrt{b^2-4ac}}{2a}`，渲染效果是这样的 $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$，使用键盘进行输入大大提升了工作效率。仿照 $\LaTeX$，我将这个图片上的公式编辑器也设计成了像 $\LaTeX$ 一样的命令式，上手程度远低于 $\LaTeX$，只需花五分钟学习就可以轻松上手，编辑效率提升许多。

另外，在设计中，这个编辑器可以对多张图片进行编辑，最后导出为 *pdf* 或长图，便于批改整套卷子而不用一份一份打开。

2 架构设计

2.1 技术选型

本论文的核心要点就是要在图片上编辑公式，那么最重要的功能必然是进行公式的渲染。目前，在计算机中进行公式的渲染大多使用 TEX [3] 。

TEX 是一套排版系统，由计算机科学家、斯坦福大学教授 $Donald\ Knuth$ 设计和编写的，于 1978 年首次发布，这是最复杂的数字印刷系统之一。TEX 被广泛应用于学术界，尤其是数学、计算机科学、经济学、工程学、物理学等等。在 TEX 的基础上，衍生出了许多封装更优雅的排版系统，例如编写这篇论文所用的排版系统 $Xe\ LaTEX$ 就是基于 TEX 封装的。

作为中学生，我们没有足够的知识储备和精力去维护一套新的渲染公式的系统，只能选择站在前人的肩膀上进行我们的项目。但是，TEX 在不同的平台上也有不同的实现，经过仔细筛查，我选用了 $Web+KaTeX$ [4] 的方案。从 $TexLive$ [5] 安装的 TEX 过于庞大而且只方便生成 pdf，并不方便直接渲染在图片上。而 $MathJax$ [6] 的速度太慢，可能会对性能造成一定影响。所以我选择了 $KaTeX$，这是一个能在网页上渲染公式的库，它的渲染速度最快，不少支持显示公式的网站也使用了这个库，例如 $OpenAI$ 的 $ChatGPT$ [7] 。

同时，使用 $KaTeX$ 支持在网页上运行，配合将文档元素转成图片的库 $html2canvas$ [8] 就可以将 $KaTeX$ 渲染出的元素转换成图片，方便在原先的图片上叠加图层。另外，使用 Web 的好处是只需要打开浏览器就可以快速编辑，免去安装的麻烦，更加人性化。如果要发行客户端，只要往上套好 $Electron$ [9] ，做好兼容层就可以。可以一次编写，到处运行。

计划将编辑器分为四个模块：总体的效果渲染，代码编辑器，当前行效果预览和图片管理。

2.2 渲染系统

公式渲染是这个设计中最核心的部分，我们的设计应该同时具备实用性和易用性，那么“所见即所得”是我们追求的目标之一，在编辑器中应该划分一部分出来区域来渲染预览效果。为了保证预览效果，预览区域的形状大小应该和画布形状大小是相似的，即预览的宽高比和原始画布的宽高比相等。用数学的方法来写，就是

$$\frac{width_{canvas}}{height_{canvas}} = \frac{width_{raw}}{height_{raw}} \tag{1}$$

这样能保证预览不会变形。
为了方便计算，我们将上式变形为

$$ratio = \frac{width_{canvas}}{width_{raw}} = \frac{height_{canvas}}{height_{raw}} \tag{2}$$

[i] [ii] [iii] [iv] [v]

[i] $ratio$: 比例
[ii] $width_{canvas}$: 预览区宽度
[iii] $height_{canvas}$: 预览区高度
[iv] $width_{raw}$: 原始图像宽度
[v] $height_{raw}$: 原始图像高度

前面已经提到一些，文本渲染的编译链如下

如果想添加其他图片、线条、形状的话可以在图层的地方合并生成预览图。

2.3 用户输入

在 *Microsoft Word* 中，输入公式只能靠鼠标单击键入，虽说 *Word* 提供了一点 LaTeX 支持，但是兼容性仍然不是很好，无法做到在正统的 LaTeX 中那么流畅，使用 *Word* 的工作效率未免有些低。所以，我们仿照 LaTeX 的设计，设计了一种脚本，也就是通过代码来帮助绘图。键入代码需要文本编辑器，在这里我选择用 *Monaco Editor* [10] 。

Monaco Editor 是微软的开源项目之一，是一个基于浏览器的代码编辑器。微软著名的代码编辑器，号称重新定义代码编辑的，*Visual Studio Code* [11] 就是脱胎自 *Monaco Editor*。

设计每帧从 *Monaco Editor* 中读取代码内容进行分析，生成错误或警告在编辑器中提示，识别当前行内容告诉预览模块显示什么内容，这样可以做到实时的效果用户体验更好。

2.4 预览

因为全部代码绘图有些许延时（比如说将元素转换成图片需要几秒时间），并不能很方便快捷的查看当前行的设置（字体/颜色/图片）会有什么效果，所以专门设置了预览区，省去合并图层的步骤更加节省时间。

2.5 图片管理

有时候需要从外部导入图片（比如说几何图形或者校徽），设计了一个区域来管理和预览图片，默认缩放到容器高度 50%，这样可以在保证能看清图片的情况下堆放更多的图片。

2.6 页面布局

计划将编辑器的左上作为输入、编辑代码的区域，右上为总体效果的预览区，左下为单行效果的预览区，右下为图片管理区。

2.6.1 布局大小计算

根据*(1)*，要保证右上的预览与画布尺寸等比缩放。不妨设页面宽度为 $width_{container}$，高度为 $height_{container}$ 。则有

$$width_{editor} = width_{container} - width_{canvas} \tag{3}$$

$$height_{editor} = height_{canvas} \tag{4}$$

$$width_{image} = width_{canvas} \tag{5}$$

$$height_{image} = height_{container} - height_{canvas} \tag{6}$$

$$width_{preview} = width_{container} - width_{canvas} \tag{7}$$

$$height_{preview} = height_{container} - height_{canvas} \tag{8}$$

对应关系如下表

表 1: 编辑器布局变量名对应表

名称	下标	内容
width	*container*	容器（页面）宽度
	canvas	画布宽度
	editor	代码编辑器宽度
	preview	单行预览区宽度
	image	图片管理区宽度
height	*container*	容器（页面）高度
	canvas	画布高度
	editor	代码编辑器高度
	preview	单行预览区高度
	image	图片管理区高度

2.6.2 画布缩放

当画布原始尺寸大于预览区大小时，我们需要对其缩放。不妨设画布原始宽度为 $width_{raw}$，原始高度为 $height_{raw}$。在计算时，首先要判断是横屏还是竖屏，如果是横屏，那么要尽可能在水平方向上缩放到最大，如果是竖屏，那么要尽可能在竖直方向上缩放到最大，能保证画布始终是最大且不会变形的。

需要注意的一点是，需要对预览区大小做一定限制，以防止其占用区域过大影响到代码编辑器和图像管理区不能正常使用。我们为左侧代码编辑器和效果预览区预留了至少

表 2: 各分区最小最大尺寸

名称	最小尺寸 (宽度 × 高度)	最大尺寸 (宽度 × 高度)
代码编辑器	$40\% \times 80$	$(width_{container} - 240) \times 80\%$
画布	240×80	$60\% \times 80\%$
单行效果预览区	$40\% \times 20\%$	$(width_{container} - 240) \times (height_{container} - 80)$
图片管理区	$240 \times 20\%$	$60\% \times (height_{container} - 80)$

$40\% \times width_{container}$ 的宽度，为下方的效果预览和图片管理预留了至少 $20\% \times height_{container}$ 的高度。

如果页面过小的时候，画布可能会被缩放的很小，所以要对此限制一下最小的尺寸为 240×80。

如果是横屏，先计算宽度。

$$width_{canvas} = min(width_{raw}, width_{container} \times 60\%) \tag{9}$$

如果原始尺寸很小就不必缩放了，否则预览的字体大小图片定位可能和最终的渲染图有偏差。

根据 (2) 那么缩放比例就为

$$ratio = \frac{width_{canvas}}{width_{raw}} \tag{10}$$

画布的高度就是

$$height_{canvas} = height_{raw} * ratio \tag{11}$$

注意到，$height_{canvas} \geq 80\% \times height_{container}$ 是可能的，会对页面布局造成影响，所以在算完后还有特判一下高度是否大于 80%。

竖屏就先计算高度。

$$height_{canvas} = min(height_{raw}, height_{container} \times 80\%) \tag{12}$$

根据 (2) 缩放比例为

$$ratio = \frac{height_{canvas}}{height_{raw}} \tag{13}$$

宽度是

$$width_{canvas} = width_{raw} * ratio \tag{14}$$

同样需要特判下宽度是否大于 60%

结合上述的计算过程，这是最终在工程上的源代码。 .

```
1  if (rawCanvasSize.width >= rawCanvasSize.height) {
2      // landscape
3      // fill width first
4      transferCanvasSize.width = Math.min(containerWidth * 0.6
           /* leave at least 40% space for editor */,
           rawCanvasSize.width);
5      transferCanvasSize.height = ~~(rawCanvasSize.height *
           transferCanvasSize.width / rawCanvasSize.width); //
           scale in the same ratio
6      if (transferCanvasSize.height > containerHeight * 0.8 /*
           leave at least 20% space for image manager and render
           preview */) {
7          // oh, it's too high
8          // we choose filling height
9          transferCanvasSize.height = containerHeight * 0.8;
10         transferCanvasSize.width = ~~(rawCanvasSize.width *
               transferCanvasSize.height / rawCanvasSize.height);
                // scale in the same ratio
```

```
        }
        if (transferCanvasSize.height < 80) {
            transferCanvasSize.height = 80;
        }
        if (transferCanvasSize.width < 240) {
            transferCanvasSize.width = 240;
        }
    } else {
        // portrait
        // fill height first
        transferCanvasSize.height = Math.min(containerHeight *
            0.8, rawCanvasSize.height);
        transferCanvasSize.width = ~~(rawCanvasSize.width *
            transferCanvasSize.height / rawCanvasSize.height);
        if (transferCanvasSize.width > containerWidth * 0.6) {
            transferCanvasSize.width = containerWidth * 0.6;
            transferCanvasSize.height = ~~(rawCanvasSize.height *
                transferCanvasSize.width / rawCanvasSize.width);
        }
        if (transferCanvasSize.height < 80) {
            transferCanvasSize.height = 80;
        }
        if (transferCanvasSize.width < 240) {
            transferCanvasSize.width = 240;
        }
    }
```

3 命令系统

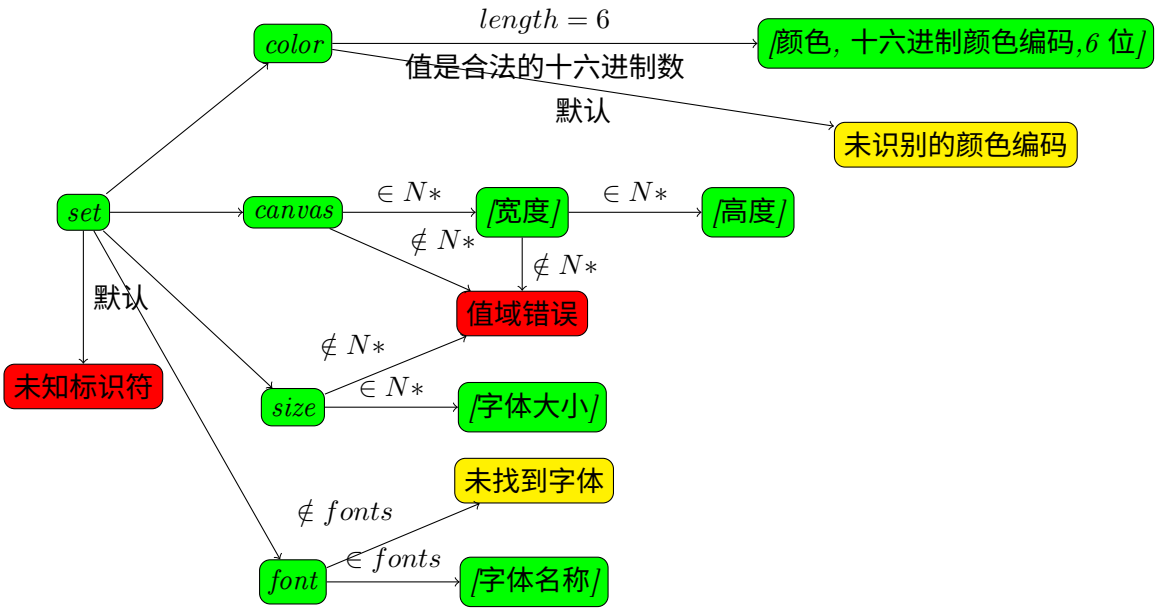
3.1 引言

像 LaTeX 一样，我们需要提供一个比较简便的命令系统（注意，这是图灵不完备的）来使用户控制画布的排版、图像的添加、文字的渲染等问题。为了方便使用，命令应该像这样：`draw from 30 30 to 300 300`，比较贴切自然语言，使得用户学习起来更方便。另外，以空格作为分隔符，在处理这些命令的时候也比较方便，性能也比较高。

3.2 命令设计

3.2.1 画布元数据

在画布中我们需要设置一些基本信息，例如画布尺寸，字体颜色，字体大小，用什么字体等等。如果在画布外面单独弄一块区域出来设置，那么在调整的时候还是得需要鼠标，会对工作效率有一定影响。为此，设置画布的基本信息应该也在命令系统中提供。既然是设置信息，不妨就将命令名称称为`set`，那么就设置字体就是`set font`，设置颜色`set color`，设置字体大小`size`，设置画布尺寸`set canvas`。可以得到以下 DFA[vi]



[vii]

宽度、高度、字体大小都必须是正整数 $(\in N*)$，颜色编码应该为 6 位十六进制数字，代表 RGB，前两位是红色的强度，中间两位是蓝色的强度，后两位是绿色的强度，值域为 $[0, 255]$(十进制) 或 $[00, ff]$（十六进制），值越大表示强度越大，占比越多。

3.2.2 定位方式

关于元素如何在画布中定位，我设计了两种定位方式。第一种是 abs，无论画布大小如何变化，元素始终会渲染在给出的坐标位置。第二种是 rwd，响应式布局，需要给定一个纯小数（百分比），具体的渲染坐标会随着画布的大小变化而变化，但与各边保持的比例始

[vi] 确定有限自动状态机

[vii] 绿色部分是合法的标识符，黄色是警告，红色是错误（会中断编译）

表 3: 常见颜色的 RGB 值

颜色	十六进制颜色编码	渲染效果
白色	$ffffff$	白底黑字
黑色	000000	黑底白字
红色	$ff0000$	红底白字
绿色	$00ff00$	绿底黑字
蓝色	$0000ff$	蓝底白字
黄色	$ffff00$	黄底黑字
青色	$00ffff$	青底黑字
紫色	$ff00ff$	紫底白字

终相同。下面是计算公式

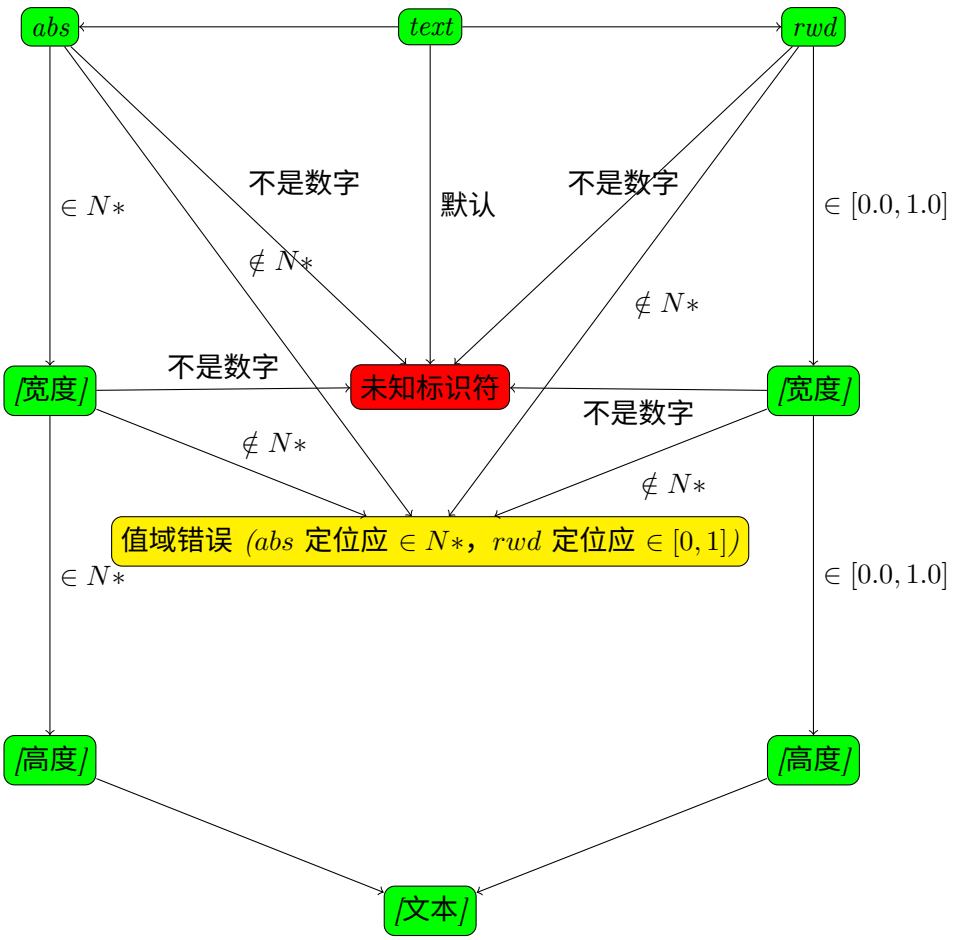
$$x_{transformed} = x_{percentage} \times canvas_{width} \tag{15}$$

$$y_{transformed} = y_{percentage} \times canvas_{height} \tag{16}$$

[viii] [ix]

3.2.3 文本渲染

结合元素的两种不同定位方式，得到以下 DFA

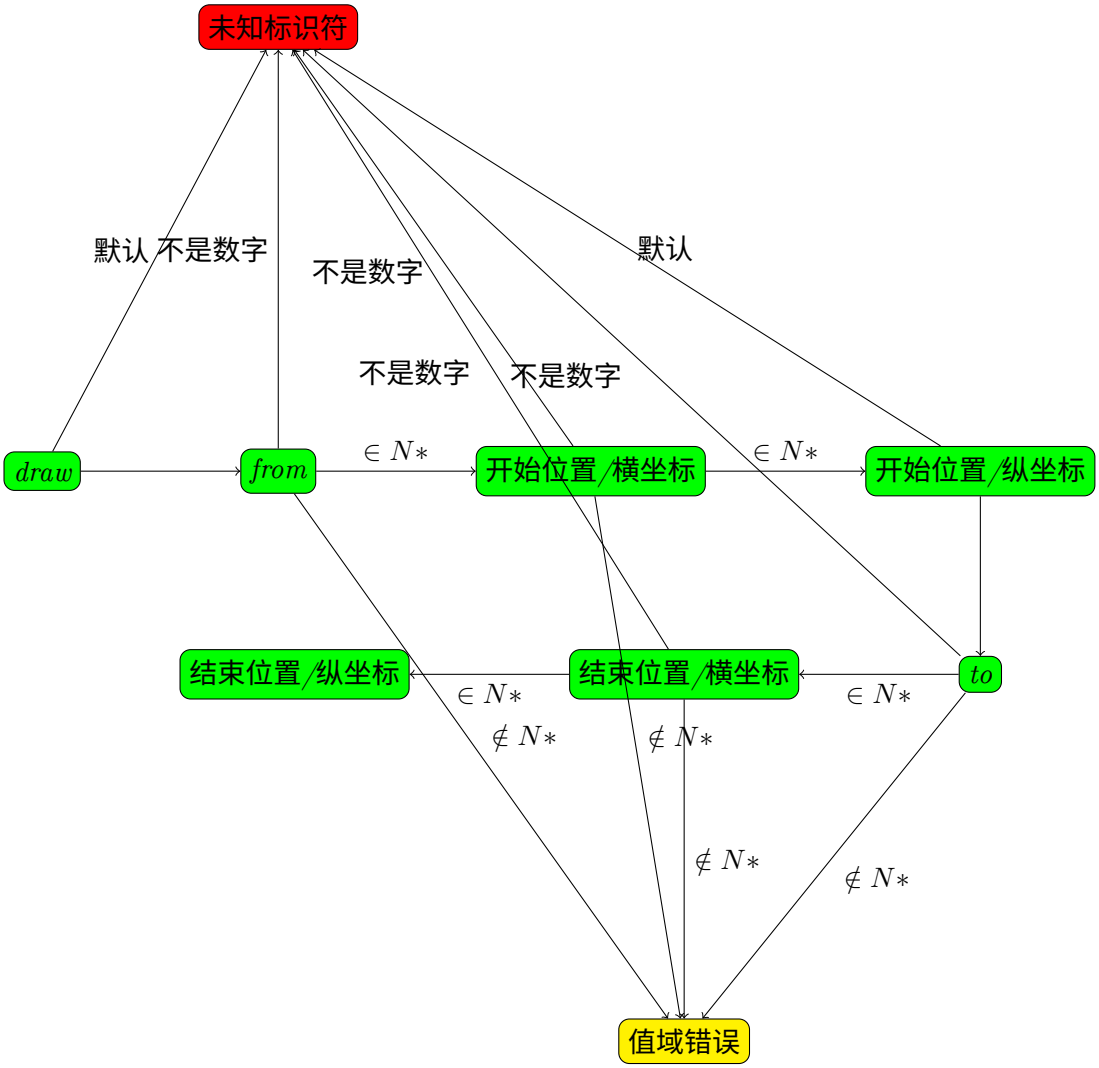


[viii] $transformed$: 已经转换好的绝对坐标

[ix] $percentage$: 元素在页面中位置的百分比

3.2.4 画线

从 (sx, sy) 画线条到 (dx, dy)，线条粗细受`set size`语句影响



需要注意一点，这里的坐标都是在 abs 布局下的，对于 rwd 布局的支持作为可以改进的部分。

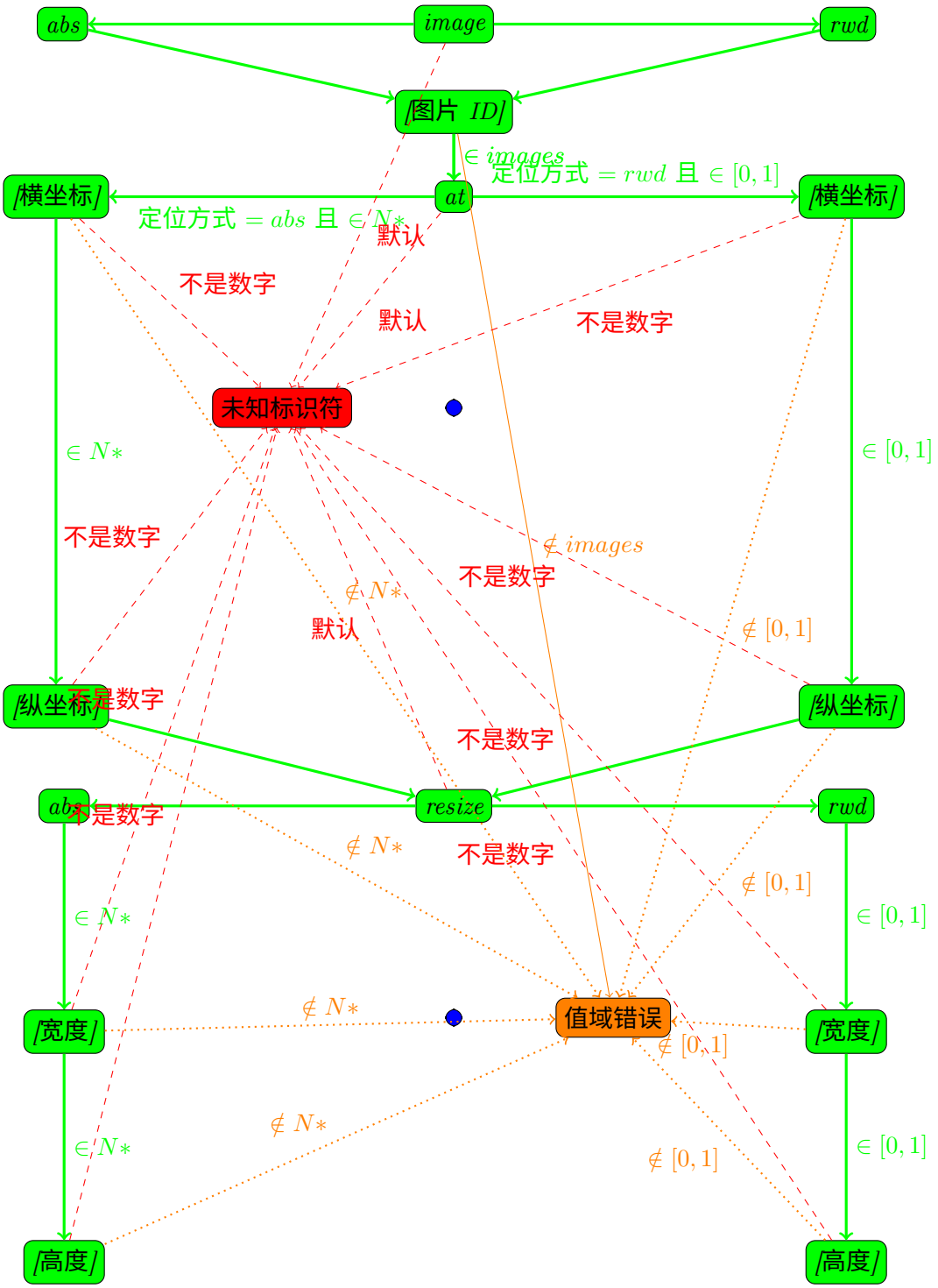
3.2.5 添加图片

有时候也需要添加一些几何图形或其他图片到画布上，命令系统也提供了支持。添加图片包括三个部分：

- 图片 ID
- 图片位置
- 图片尺寸

命令格式如下 $image\underbrace{abs/rwd\overbrace{[sha256]}^{ID}at[x][y]}_{location}\underbrace{resize[width][height]}_{size}$ [x] [xi] [xii] [xiii]

根据上面的命令格式，可以生成一张很大的 *DFA* 图：



[x] $image$：图片命令
[xi] ID：图片 *ID*
[xii] $location$：图片定位
[xiii] $size$：图片尺寸

3.2.6 LaTeX 宏

公式的环境是 LaTeX 的数学环境，有时候公式中有大量重复的命令，输入繁琐不方便，就可以设置一个简单的宏来代替。比如用 .

```
\root-1o2
```

来代替 .

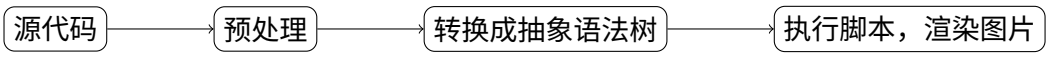
```
x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}
```

上面例子的命令就是 .

```
macro \root-1o2 x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}
```

命令定义为：macro 宏名 宏值

3.3 编译过程



先预处理，转换宏，做好行号标记，方便后面在编译过程中输出报错行号。在编译过程中，通过空格切割标识符，剔去多余的空格，根据上节中的状态机将代码转换为 AST。[xiv]

最后再根据 AST 中语句的类型进行解释运行。

[xiv] AST: 抽象语法树

4 技术难点和性能优化

4.1 技术难点

4.1.1 多语言支持

为了给全球的用户提供方便，我将默认的语言设置成了英语。但是，我在中国，而中文作为世界上使用人数最多的语言，应该也要提供对中文的支持。所以我做了多语言的支持。

具体的方案是在不同的语言环境（设备默认语言，用户设置语言）下显示不同的文字，就可以考虑在源文件只保存标记，比如`i18n.hello`，在加载的时候，根据用户语言的不同，加载不同的语言文件，替换掉原来的标记。比如在`en-US`环境下，可以有`hello: "Hello"`，在`zh-CN`环境下，可以有`hello: "你好"`。这样子可以做到不同语言环境显示的语言不同。

但是，有时候，要渲染的文本中有其他的数据，比如说以下语句：`You have 10 seconds left.` (`en-US`)；`你还剩10秒时间` (`zh-CN`)。英语和汉语的表达方式和语序有些差异，不能单纯这么拼接。

这时就要考虑使用模板插值，在语言文件中把要填的数据写成标记，在根据语言渲染文字的时候获得数据，把标记替换成数据，这样就能够保证语序正常又能显示数据。按上面的例子，英语的语言文件中可以这么定义：`test: "You have [[ time ]] seconds left."`，中文的语言文件中这么定义：`test: "你还剩[[ time ]]秒时间。"`，这中间的`time`就是插值的标记，渲染的时候就传入数据 .

```
{
    "time": 10
}
```

替换掉标记。

可以考虑使用正则表达式匹配标记，例如上文的`[[ ]]`就用 .

```
/\[\[.+?\]\]/g
```

最外面的一层`/ /g`是声明这是个正则表达式，全局匹配。`\[\[`匹配前两个方括号。中间的`.`表示匹配除换行符外的一切字符，`+?`表示匹配数量一个及以上，尽可能少匹配，否则有多个插值的话就会匹配成第一个左方括号到最后一个右方括号。`\]\]`匹配后两个方括号。

为了让代码更美观一点，方括号之间的空格可写可不写，我们就需要把匹配出来的插值标记的值删去开头两个、末尾两个（去掉方括号）后清除掉前后的空格。最后根据提供的数据替换文本就可以了，下面是最终的源代码。 .

```
template(key, arg) {
    let sourceString = map[key];
```

```
        const templateMatchRegExp = /\[\[.+?\]\]/g;
        const templates = Array.from(sourceString.matchAll(
            templateMatchRegExp));
        templates.forEach(match => {
            const src = match[0].slice(2, -2).trim();
            if (arg[src]) {
                sourceString = sourceString.replace(match[0], arg
                    [src].toString());
            } else return;
        });
        return sourceString;
    }
```

4.1.2 多行文本命令

上文提到，命令处理的时候是按行切割的，这意味着文本命令也只能写在一行内，中间通过\\进行换行。对于换行多的文本编辑、阅读起来就很不方便。为此，我设计了一个宏，支持多行文本命令的编辑，在预处理阶段会将多行文本转换成一行进行编译。如下 .

```
    text rwd 0.5 0.5 `
    Hello World
    $x^2+y^2=z^2$
    `
```

等同于

```
    text rwd 0.5 0.5 Hello World $\\$ $x^2+y^2=z^2$
```

渲染效果如下

Hello World

$x^2 + y^2 = z^2$

可以考虑使用 *DFA*，先扫描把在`之间的宏标记出来，如下图 .

将状态在”多行字符串宏”的字符记录下来，就是匹配出的宏。对于每个宏，可以用正则表达式将宏中所有的换行符替换成\\标记。

但是，经过预处理后的代码行数会与源代码不一样，按原来的方式，在编译过程中产生的错误的行数会产生偏差，无法映射到正确的行上。所以考虑预处理时，在每一行的末尾标记上原来的行号，编译器编译的时候只需要读取原来的行号作为错误抛出就行了。具体代码请见下表。

- 预处理：https://github.com/lihugang/mep2/blob/master/src/utils/AST/preprocess.ts#L47-L115
- 获取预处理中标记的行号：https://github.com/lihugang/mep2/blob/master/src/utils/AST/preprocess.ts#L16-L45
- 预处理时模拟 *DFA* 的文件流实现：https://github.com/lihugang/mep2/blob/master/src/utils/simpleStringStream.ts#L35-L131
- 编译中识别源文件行号：https://github.com/lihugang/mep2/blob/master/src/utils/AST/converter.ts#L125

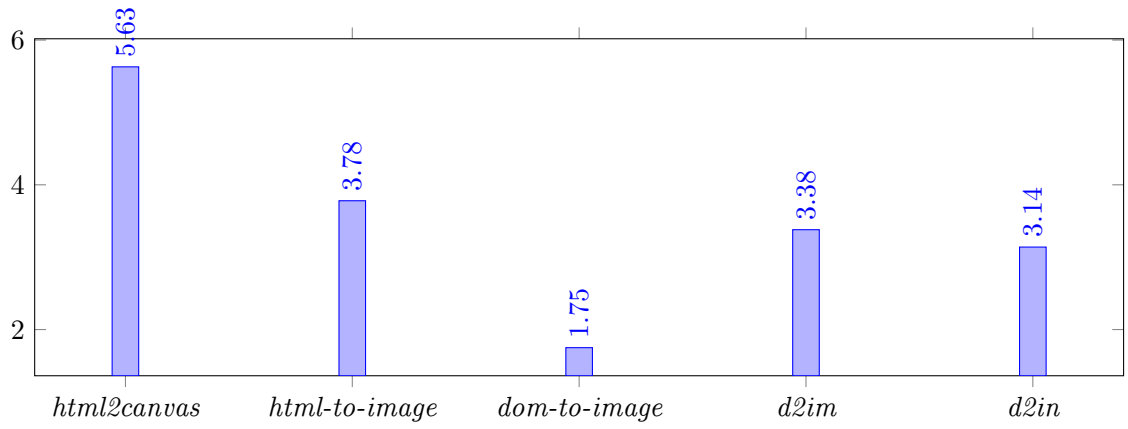
4.2 性能优化

4.2.1 数学公式渲染的性能优化

渲染数学公式时通常会导致 *3-5* 秒的卡顿，严重影响用户体验，本段只要阐述如何优化渲染数学公式时的性能。 .

渲染库

由于页面存在大量的元素（文本编辑器有很多定位），*html2canvas* 在拷贝分析源文档中花费了大量的时间。我尝试使用其他的库将页面元素渲染成图片，下图是测试结果。 .



[xv] [xvi] [xvii]

可见还是 *html2canvas* 的性能最好。 .

线程池

既然无法提升渲染速度，那就试图将渲染内容放到别的地方去，不阻塞用户操作，同时新的页面也没有那么多元素，能提高复制解析的速度。

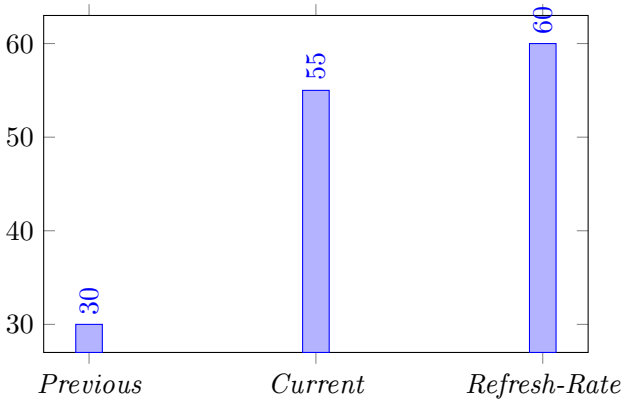
[xv] 库名字过长无法渲染，请见脚注

[xvi] *d2im: dom-to-image-more*

[xvii] *d2in: dom-to-image-next*

考虑线程池，可以请求用户弹出几个窗口，将渲染任务轮流分配到这几个窗口上，如果窗口全部无法打开再在主线程中渲染。

源代码请见https://github.com/lihugang/mep2/commit/17668e040da9d8c776d5d3b872f3745567bf2f0c .



如上图，在启用渲染线程池后，*fps* 有很大提升，从 *30* 提升到 *55*，逼近屏幕刷新率 *60*。

5 展望

经过几个月的努力，我们完成了在图片上编辑数学公式的编辑器的基本设计，功能完整能正常使用。但由于人力和精力原因，一些功能还没被设计和实现，例如，集成 $ChatGPT$ 和 $MidJourney$ 智能分析生成代码和图片，画笔和橡皮擦的支持，播放模式便于授课，图片拖拽自动生成代码，版本控制与多人协作，暗黑模式，对手机端的适配等等。期待在未来能有更多的仁人志士一起去完善，提升学生、教师、科研工作者的生产力和工作效率。

致谢

感谢我的指导老师郝赛赛老师，在本论文的撰写中帮助许多。感谢 *Donald Ervin Knuth*，发明了 TEX 能让我在这的基础上设计公式的渲染，也让我能用如此方便的工具完成我的论文。最后感谢我的父母对我的养育、陪伴、理解和支持，一路走来谢谢有你们，我成长匪浅。

参考文献

[1] *Microsoft. Write an equation or formula. [DB/OL]. (2023-03-11)[2023-03-11].* https://support.microsoft.com/en-us/office/write-an-equation-or-formula-1d01cabc-ceb1-458d-bc70-7f9737722702

[2] *The LaTeX Project. LaTeX –A document preparation system. [EB/OL]. (2023-01-04)[2023-03-11].* https://www.latex-project.org/

[3] *Wikipedia. TeX - Wikipedia. [DB/OL]. (2023-03-10)[2023-03-12].* https://en.wikipedia.org/wiki/TeX

[4] *KaTeX. KaTeX - The fastest math typesetting library for the web. [EB/OL]. (2023-03-12)[2023-03-12].* https://katex.org/

[5] *TexLive. Tex Live. - Tex Users Group. [EB/OL]. (2022-02-24)[2023-03-12].* https://www.tug.org/texlive/

[6] *MathJax. MathJax | Beautiful math in all browsers. [EB/OL]. (2023-03-12)[2023-03-12].* https://www.mathjax.org/

[7] *OpenAI. ChatGPT. [EB/OL]. (2023-02-13)[2023-03-12].* https://chat.openai.com/chat

[8] *html2canvas. html2canvas - Screenshots with JavaScript. [EB/OL]. (2022-01-22)[2023-03-12].* https://html2canvas.hertzen.com/

[9] *Electron. Build cross-platform desktop apps with JavaScript, HTML, and CSS. [EB/OL]. (2023-03-12)[2023-03-12].* https://www.electronjs.org/

[10] *Microsoft. Monaco Editor. [EB/OL]. (2023-02-14)[2023-03-12].* https://microsoft.github.io/monaco-editor/

[11] *Microsoft. Visual Studio Code - Code Editing. Redefined. [EB/OL]. (2023-03-12)[2023-03-12].* https://code.visualstudio.com/

附录

源代码及工程实现

对于此设计，我们给出了工程上的实现，源代码存储在https://github.com/lihugang/mep2，开源协议是*The GNU General Public License v3.0*，我们已经将工程实现作为软件发布，官网是https://mep2.lihugang.top，提供了在线的演示版本，https://demo.mep2.lihugang.top，但由于浏览器的限制，一些功能可能无法使用或不能正常工作。欢迎您给出宝贵的建议。