Investigative Research into impriving image text generation

1. Pre-cache technique that sends a http head request while user is typing to server and a get request when the user submits the modified text.
   1. Problems with this technique:
      1. How do we decide when to send the http head request? It’s not good to send a request after every key press since it will make the browser send too many http requests to the server.
         1. Is it possible to have the latest head request cancel the previous ones?
      2. What if the user presses enter right when he finishes typing? Then the issue with the slow generation will still be there.
2. Measure user experience
   1. What is the requirement for a good user experience?
   2. Methods
      1. Using log file
      2. Measure speeds of:
         1. Image delivery time
         2. imagemagick
            1. Measure sentiment speeds for imagemagic/rmagick

Usually around .3-1 second depending on how long the sentiment is. The longer the sentiment, the longer it takes to generate the image because of the way it tests to see if the sentiment fits in the text box.

* + - 1. Pango/cairo
         1. Measure content delivery speeds for pango/cairo
      2. How can we get the most accurate measurements?

1. Server-Side latency improvement.
   1. Can we improve the speed of imagemagick by compiling it with different options?
   2. Can we implement a kind of pre-fork mechanism?

Curl generateTextLayer to twosmiles.com

real 10m16.489s

user 0m5.536s

sys 0m3.167s

Pango cairo (Image/Text libraries)

1. Run as an executable on OSX using node.js
   1. “time for x in {1..1000}; do curl -s "http://localhost:1337/" > /dev/null;done”
      1. real 0m11.181s
      2. user 0m4.030s
      3. sys 0m2.355s

Installing pango/cairo on Ubuntu

1. Install the required packages using apt-get
2. Link the packages in the bash profile
3. Put fonts in /usr/share/fonts
4. Executing binary file using node.js
   1. Local server curl
      1. real 0m54.306s
      2. user 0m2.220s
      3. sys 0m7.172s
   2. Remote server curl over hp network
      1. real 3m34.851s
      2. user 0m4.093s
      3. sys 0m2.098s
5. Node.js addon
   1. Local server curl
      1. real 0m36.944s
      2. user 0m2.252s
      3. sys 0m7.008s
   2. Remote server curl over hp network
      1. real 2m43.128s
      2. user 0m4.111s
      3. sys 0m2.120s

Test image

Alignment Font Size X Offset Y Offset Height Width

message Center 16 182 577 346 1137

Sentiment 1 :

Here’s a little gift to let you know that you’ve been on my mind and in my heart. I hope everything is going well with you.

Sentiment 2 :

You're in my thoughts and prayers during this tough time. Here's a little gift to lift your spirits.

Sentiment 3 :

I'm so appreciative of everything you do. Here's a little token of thanks.

1. Problem
   1. Imagemagick/Rmagick is slow
      1. Calculating bounds takes a lot of cpu time.
      2. The way it is currently implemented is to regenerate image to check bounds of text within constraints.
      3. Takes a long time to reflow the text
2. Is there a better way?
   1. Using a different image rendering library.
3. Pango/Cairo
   1. Cairo is an image rendering library
   2. Pango is the text rendering library. It creates text layouts to be rendered to an image using cairo.
4. Why is pango/cairo better?
   1. We can get the height and width of resulting text before rendering the image.
      1. This way, we can do quick comparisons to check if height and width fit within our constraints.
   2. We don’t need all the features that imagemagick provides (it’s bulky)
5. Writing the pango/cairo application
   1. Writing the application itself was pretty straight forward.It’s C++
   2. I modified example code to take in text data such as:
      1. Font, font size, the text we wanted to print, etc…
   3. This outputs a png image file that is ready to be transmitted to the viewer.
6. How do we transmit the image?
   1. I wrote, with brian’s help, a simple node.js server that executes my pango/cairo application, and sends the resulting image to the requesting client.
   2. However, there are a couple issues with this method.
      1. You need to write the image to disk before you are able to send it, which isn’t good.
      2. The application is executed every single request. This takes longer due to memory allocation times.
   3. Tests show that this is very efficient
7. Node-addon
   1. Node-addons are loaded under node as a module.
   2. Node addons are a way to write c++ programs as a dynamically linked shared object using v8, which is a library that is used for interfacing c++ with javascript
   3. Using this, I wrote a wrapper for the C++ function I created that has the ability to generate this image.
   4. This wrapper exposes the generate text image function which can be called whenever we want to generate a new image.
   5. This function is then exposed with the node-addon wrapper so that it can be called in nodejs.
8. How to handle the image once it is generated?
   1. Send it back as a buffer.
   2. Then inside nodejs, we just write it out as a png.
   3. Problem solved. No image is written to disk, and the program will stay loaded in memory as a node-addon
9. Reflowing text
   1. As brian has mentioned in his email, I hardcoded the text to be resized 20 times everytime to make sure that the resizing has no significant impact on the generation times. Turns out it doesn’t affect the generation times very much at all.
10. Demo the app
11. I ran a couple of tests:
    1. How to test:
       1. Curl the image 1000 times in a row and time it.
    2. Node-addon
       1. real 2m43.128s
       2. user 0m4.111s
       3. sys 0m2.120s
    3. executable
       1. real 3m34.851s
       2. user 0m4.093s
       3. sys 0m2.098s
12. So how does this compare to imagemagick?
    1. Same as before, curl 1000 times and time it. With the same text
    2. Imagemagick
       1. real 10m16.489s
       2. user 0m5.536s
       3. sys 0m3.167s
    3. Node-addon
       1. real 2m43.128s
       2. user 0m4.111s
       3. sys 0m2.120s
13. Next steps

Hi everyone, thanks for coming to my fast text generation demo.

For the past couple weeks, I worked investigating a different and faster way of generating text on the fly that gives users a better experience.

The problem was that image magick takes a long time to resize text into the constraints of a text box. Im can only check if the text is too big after generating it into an image and then checking it’s metrics. If it is too big, it is resized by decreasing the point size by one until it fits. This takes a long time. And if there are a lot of these images being created on 1 page, it will take a very long time.

To solve this problem, I went with a different text image rendering library called pango cairo.

Cairo is a image rendering library and pango is the font rendering library that interfaces with cairo pretty smoothly by creating layouts that cairo can use to create these images.

It’s better because we can check the bounds of the text before rasterizing it. Therefore we can do way more checks within a shorter amount of time.

So the implementation is pretty straight forward. It is written in C++.

As you can see, we create a layout and using a font description, we can update the layout’s font size relatively quickly. Afterwards, we can just call a function to get the height and width dimensions. We can loop through this many times without a significant impact on performance.

There were a few short comings with this c++ application.

This program is executed on every request. And the program writes every single image to disk, which is very inefficient because of disk read and write times!!

I rewrote the program as a node-addon. This way, the program can stay loaded into node.js’s application memory, and we can also just directly send the image buffer back to the client.

I used a library called v8 which interfaces C++ with javascript, allowing us to expose my text generation function.

As you can see, the node-addon performed better than the executable after 1000 requests.

The node addon returns the image buffer to nodejs server, then nodejs writes it out as a png.

Problem solved.

Demo.

Wrote a quick demo site to show the results.