TOOLS FOR RELATING TIME AS IMAGE AND IMAGE AS TIME

IN

GRAPHICAL

TRANSLATION

CPSC 491 Senior Thesis

Fall 2021

In Graphical Translation

Tools for relating type as image and image as type.

By Felicia Chang

Supervised by Holly Rushmeier and Julian Bittiner

Introduction

Design software keeps things simple. It is easy to arrange and rearrange a collection of type and image. Combine the two, and things get more complicated. Design professors repeat the refrain: type is image. But is it? The methodology to interpolate between image[1] and text is sparse.

I am interested in design software that doesn't keep things simple. My thesis is an attempt to blur the lines of legibility and hierarchy to achieve a unity between type and image.

My design professor, Pamela Hovland once described type as a container. What information can typography hold besides content? Emotion is typically the immediate answer. Imagine same word written in a thin serif as opposed to a bold italics sans serif.

What about visual context? That is, can we derive a more direct relationship between type and the space in which it sits? How can they speak the same visual language? These questions culminated in two image-making tools: one that uses an image to build a typeface and another that manipulates a typeface to match the image. Both tools seek to translate between the two elements we use most in design: type and image.

---

[1] Image in this usage is literal. Here, I'm talking about pictures.

Background

In my junior summer, I started exploring the weird web – designers who were doing wildly experimental web-based work. Objects and text moved in unexpected yet poetic ways.

In design classes, we were taught to make work that was static. I was interested in interpreting that static work in an interactive way. As an amateur writer and experienced engineer, I built a website called Sine Qua Non. It was a collection of interactive poems I created using ReactJS and P5JS.

The next year, I created an editor called Alibi that gave programmers the ability to publish their own interactive poetry. The editor was not fully realized, but the desire to build tools for digital expression remained.

I was drawn to interactive poetry because it introduced new ways to read, one that was visually playful and exciting. A poem could feel choppy and so could the formal qualities of the text. And because it was programmatic, those formal qualities didn't need to be static.

Here began my fascination with expressive typography. One can imagine expressive typography as hieroglyphics. Words are distilled into contextualized images. Illustrative typography is also expressive. The typography is hand drawn to convey emotive qualities. Expression and information in typography is a never ending exploration.

My chapter of that exploration lies in the digital realm. I am interested in the computer's interpretation of our world, not because computers have any sentient thought but because its abstracted processes are opportunities for surprise and experimentation. In my thesis, I harness the analytical ability in machine learning and computer vision to create visual work. Sometimes, the results are beautiful. Sometimes they aren't. The variability has become part of my process.

Methodology

Period of Exploration

1. Contextualization

I entered my thesis work with a commitment to exploration. I had vague ideas about how I wanted to relate type to image, but nothing was set in stone.

I looked to three creative technologists for inspiration: Zach Lieberman[2] from the MIT Media Lab, Yuin Chen[3] from the Google Creative Lab, and Sascha Lobe[4] from Pentagram. From Zach Lieberman, I saw how images could be used to alter text to blur the line of legibility. From Yuin Chen, I saw how text could be created computationally through the use of a variable grid. From Sascha Lobe, I saw how geometric text acted as a welcome reprieve against complex background images. Before me were various modes of utilizing text and images, all of which brought themselves into my work.

2. Focused Building

For the months of September and October, I focused on preliminary prototypes for two ideas I had. For September, I developed a way to translate 2D images into a 3D models that I could project text onto. For October, I learned how to extract shapes from an image to create type with. While neither tool was polished by the end of the month, each tool did something. That something though, was brittle.

Period of Iteration

Once I had a proof of concept, I extended the tools' usability. Type attributes could now vary by user input, such as size, weight and shape. The display of the images and text are variable too. Feature development was spurred by my own application of these tools. It involved noticing limitations to my tool and subsequently extending its usability.

---

[2] http://zach.li/

[3] https://yuinchien.com/

[4] https://www.pentagram.com/about/sascha-lobe

Technology Overview

1. Image Processing

I utilized Python libraries such as OpenCV[5] and PyTorch[6] to conduct image processing. In the sections below, I will go into more detail about how each library was used to achieve my desired results.

2. Image Manipulation

For image manipulation and display, I used the Javascript graphics libraries, WebGL[7] and P5JS[8]. I chose to use WebGL for lower level graphical needs and displaying 3D objects. I used P5JS for simpler graphical tasks like image display and arrangement, as it abstracts away much of the boilerplate needed for WebGL.

3. GUI

I used Flask[9], a web framework built as a Python module. Using Flask, I was able to easily connect the image processing work done in Python with Javascript. My Flask app was divided into three parts: the root Python file (app.py), templates that contain HTML, and scripts that run Javascript.

---

[5] https://opencv.org/

[6] https://pytorch.org/

[7] https://get.webgl.org/

[8] https://p5js.org/reference/

[9] https://flask.palletsprojects.com/en/1.1.x/

About

Type as Image is a tool that extracts shapes, particularly rectangles and circles from images to create typefaces. It utilizes image processing tools from the OpenCV library to extract shapes. Then, using a Javascript graphics library, P5JS, it builds letterforms from the shapes. This type of tool is inspired by variable typefaces that are endlessly generative.

Technology

1. OpenCV

To extract shapes from images, I used OpenCV's image processing API. I focused on extracting rectangles and circles, given their frequent appearance in letterforms.

To identify rectangles, I experimented between the Hough Line algorithm and Template Matching algorithm. Template Matching was more successful, as the Hough Line Algorithm made delineating the bounds of the rectangle difficult. Extracting circles with the Hough Circles algorithm was successful.

2. P5JS

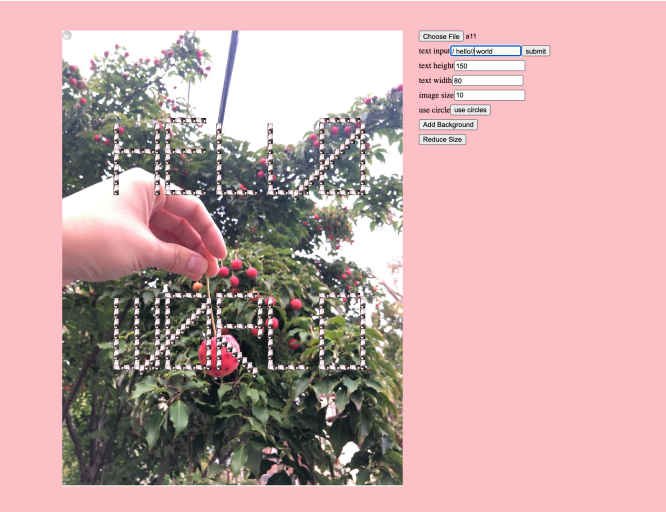
Once I had my rectangles and circles on hand, I had two choices. I could either create letterforms using P5JS's built in textToPoint function, which turns TrueType letterforms into a set of points. Or, I could create the letterforms from scratch. I chose the latter for more organic, childlike characteristics. To build the letterforms from scratch, I reduced each letter to a set of rectangles and circles that either run vertically or horizontally. I then created a series of helper functions that arranged the rectangles and circles given a set of variables ie. Input text, letter width, height and image size.

3. Glue between Python and Javascript

To send images between Python and Javascript, I created an editor that gives users a number of inputs. Those inputs get passed to the Python script, which returns an extracted rectangle and circle. P5JS handles the subsequent rendering work.

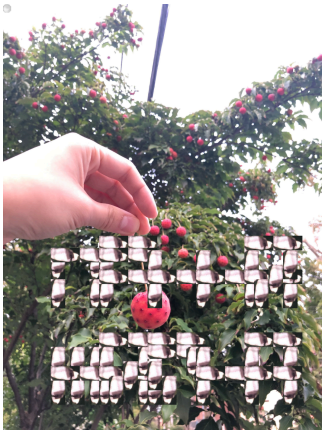
Editor

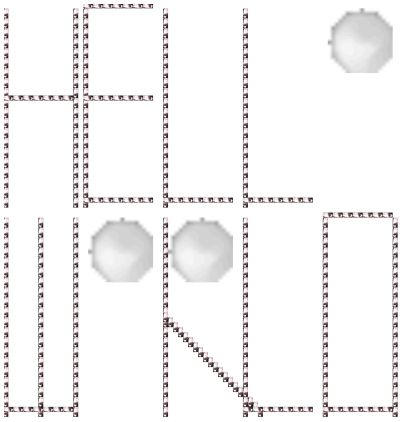
The editor was created in response to variables that I wanted access to.



On the right had side, there are a number of inputs that are malleable. The image to the left is created with the following:

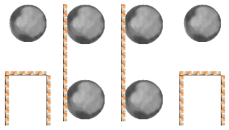
input text = "hello world" (slashes are used to denote line breaks).
text height = 100 pixels
text width = 80 pixels
cropped rectangle = 10 pixels in width.
use circles = false
add background = true.

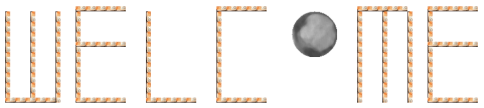




Advantages

With just a few variables, I can create endless variations in form. Another example is show below:





The degree of separation between the typography and its original image is left to the user. Though my initial goal was to create a tighter link between image and text, the typeface is able to function without the image as well. This is a testament of the unique quality of the letterforms created with simple shapes and some experimentation.

The editor encourages exploration. Tweaking a variable slightly can create dramatically different results. At the same time, this editor rewards experience. Once you know how the tools work, you can achieve your desired results.

Disadvantages

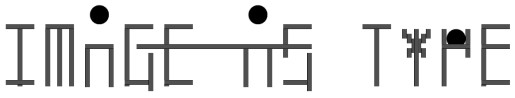
The program's variability does have a limit. The fundamental structure of each letterform is created with a grid in mind. Each letter is broken into its simplest representation, like bitmap fonts. These letterforms therefore take on a "glitchy" aesthetic.

There is a certain "black box" nature to the letterforms. Though rectangles and circles are sourced from the sample image, it can be hard to identify where exactly the shape was found. Users also don't have control over which rectangles and circles are chosen. Sometimes, I have found myself suggesting shapes to the program by cropping out the background. Such preprocessing shouldn't be needed.

Future Development

While the current set of variables already allow for a breadth of possibilities, there are even more choices a user can have. For example, one might want to change the space between each letter, adjust the background image or create animation loops.

And though I like the idea of having the computer choose rectangles and circles for me, giving greater user control can also be a boon at times. Presenting alternative choices is important in creating a dynamic tool.



About

Image as Type is a typographic projection tool. 2D images are converted to 3D models using a Monodepth machine learning library. Text can then be projected onto a sequence of images, creating a grid or collage of rotating models.

Technology

1. Monodepth

Rather than spend my time building my own machine learning model to extract depth from images, I used a monodepth library[10] that used Pytorch to train a depth estimation model by implementing the method described in the paper, “Digging into Self-Supervised Monocular Depth Prediction.”

After finding the depth values for each pixel using the monodepth library, I created an OBJ file to document the x, y and z values for each pixel. In the process of

[10] https://github.com/nianticlabs/monodepth2

documenting the position of the pixel, I recorded the RBG values as well. I then “projected” text onto the image by changing RGB values where text was present.

2. WebGL

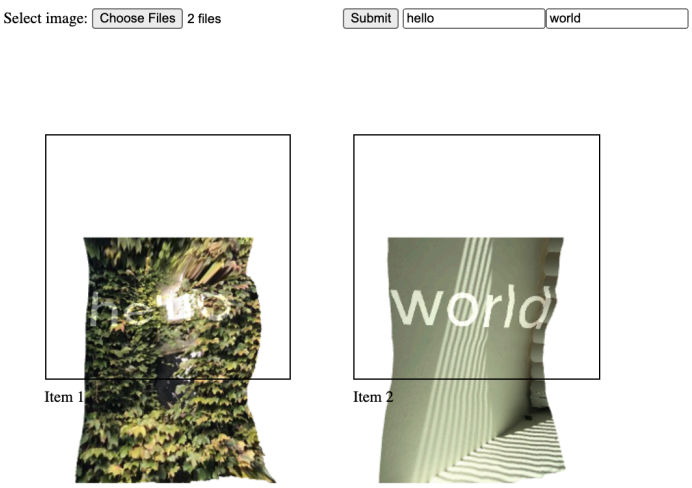
To display the 3D model of a given image, I needed a Javascript library that could handle 3D rendering. I wanted a graphics library a bit more low-level than P5JS or ThreeJS. WebGL was a good fit. Using WebGL, I read the OBJ file as buffer data. To model multiple OBJ files, I simply created an array of objects, each object representing one model.

3. Flask

Like the other tool, I used Flask to glue the machine learning and graphics work together. Users provide can provide multiple image with corresponding input text. That information is then passed to the Python code and processed together.

Editor

In this editor, the user acts as a curator. Their job is to collect image-text combinations that create a story of their own choosing.



As a result, the variables that one can choose sparser: you simply provide the image files and corresponding text.

The objects in the canvas also rotate, additional screen captures can be seen below.

Advantages

The ability to make multiple canvases, as indicated by the item marker on the bottom left, gives this editor the potential to create interesting collages. I was interested in challenging the legitimacy of associations – how does this word work with this image. And, how do they all work together when arranged together.

The typography is manipulated to take on the contours of the image. The image itself no longer looks like its original. Both the images and text are processed into new forms.

Disadvantages

While the 3D forms of the image are intriguing to view, they are not accurate representations of the contents of the image. The original intention when building this tool was to transform the 2D image plane into 3D representations of the objects within the image. However, such a machine learning application does not exist for granular detail – the library I used was meant for self-driving cars which only need vague interpretations of its environment. I did look into other libraries, such as Nvidia's PlaneRCNN[11]. These libraries require more robust GPU's than available on a laptop, unfortunately.

Future Development

While the fundamentals of this tool is simple, there are ways in which we can add more customizability. There are two areas that can be improved:

1. Typographic presentation

    Currently there is only one typeface option because Python accesses typefaces through ttf files. Having greater freedom of choice over which typeface to use is in itself an area of exploration.

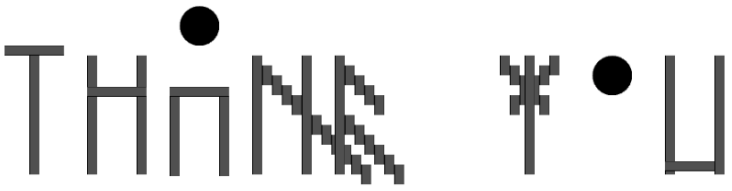
    Additionally, the text presentation is not smart, meaning it cannot adjust to the amount of text. If the number of characters exceeds the image width, it will bleed over.

---

[11] https://research.nvidia.com/publication/2019-06_PlaneRCNN

2. Image arrangement

The image grid follows the order in which the images are added. I would like to have greater flexibility over the order and the corresponding text image pairings. I didn't implement this feature for my prototype due to the amount of re-rendering the would be required. To do this work well, I would have to think of more efficient ways of making these images malleable.



Acknowledgements

I would like to thank my advisors, Professor Holly Rushmeier and Professor Julian Bittiner for their support, advice and supervision – all of which were very much needed for the fruition of my thesis.

I would like to also thank my professors this semester, Pamela Hovland, Verlyn Klinkenborg, Alexander Valentine, and Chris Pullman for stretching my creative practice in ways I would not have imagined.

Lastly, many thanks to Sarim Abbas for paving the way before me and Jack Adam for being my fellow CPAR major. Thank you to my family for their endless support.