Algorithmic Layout of Characters in Perspective

Mariel Bass[1], Erik D. Demaine[2], and Martin L. Demaine[2]

[1]Glass Lab, Massachusetts Institute of Technology, Cambridge, MA, USA; mariel@marielbass.com
[2]CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA; demaine@mit.edu

Abstract

We describe a simple and practical algorithm for arranging a collection of images ("characters") in a perspective layout that looks uniform. Randomness and tunable parameters make for varied layouts, with no repetition of a pattern. We present an application of this algorithm to an art installation made from paper printed with computed layouts and folded along curved creases.

Introduction

In this paper, we describe the making of "Hanging Out", shown in Figure 1: an installation of twelve pieces of paper, each printed with a unique algorithmic layout of hand-drawn characters, folded along curved creases, and hung from the ceiling (in addition to a few pieces of paper on the floor). The total number of characters in the hanging pieces is 41,732, the 2021 population of Fitchburg, Massachusetts, where we did the installation.

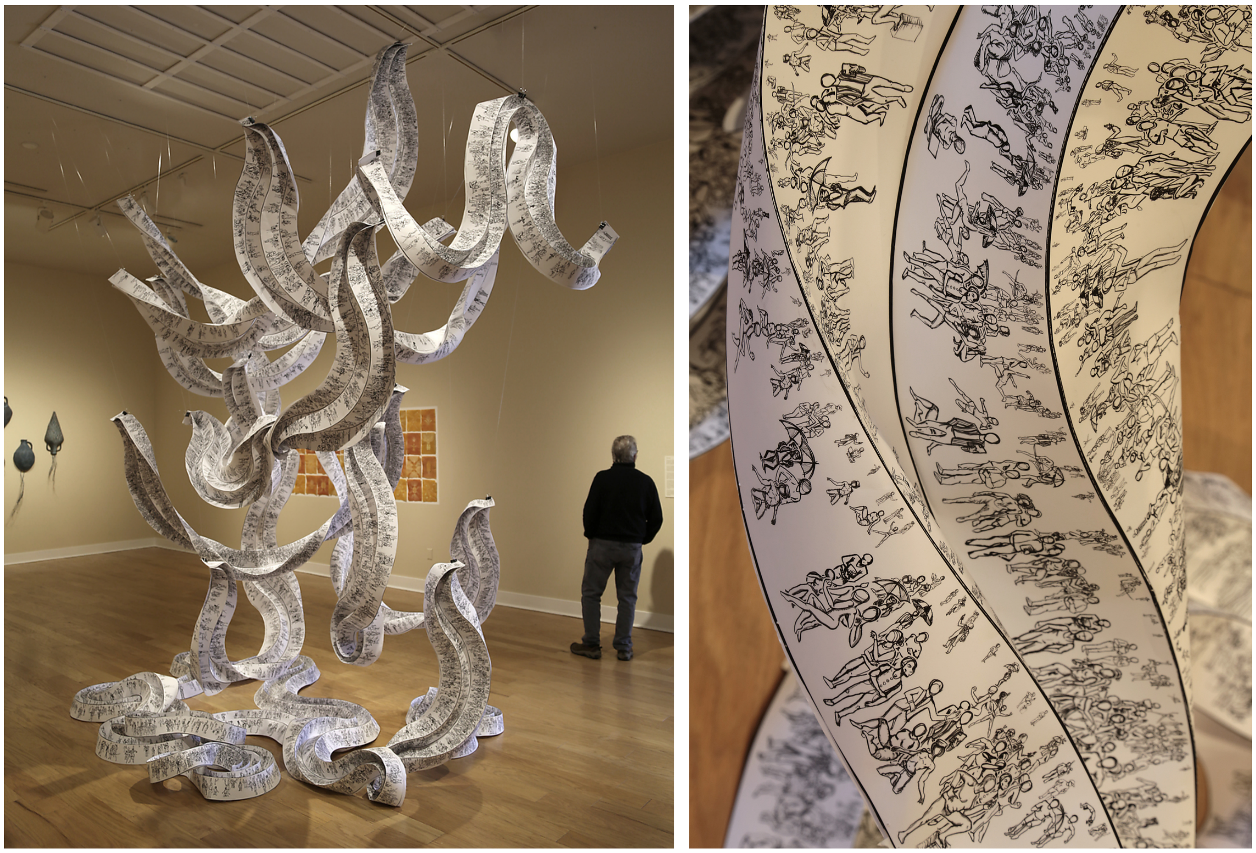


Figure 1: *Photographs of "Hanging Out" installation at Fitchburg Art Museum, 2023. Printed and folded paper, 5ft × 10ft × 14ft tall. Sculpture and photographs by the authors.*

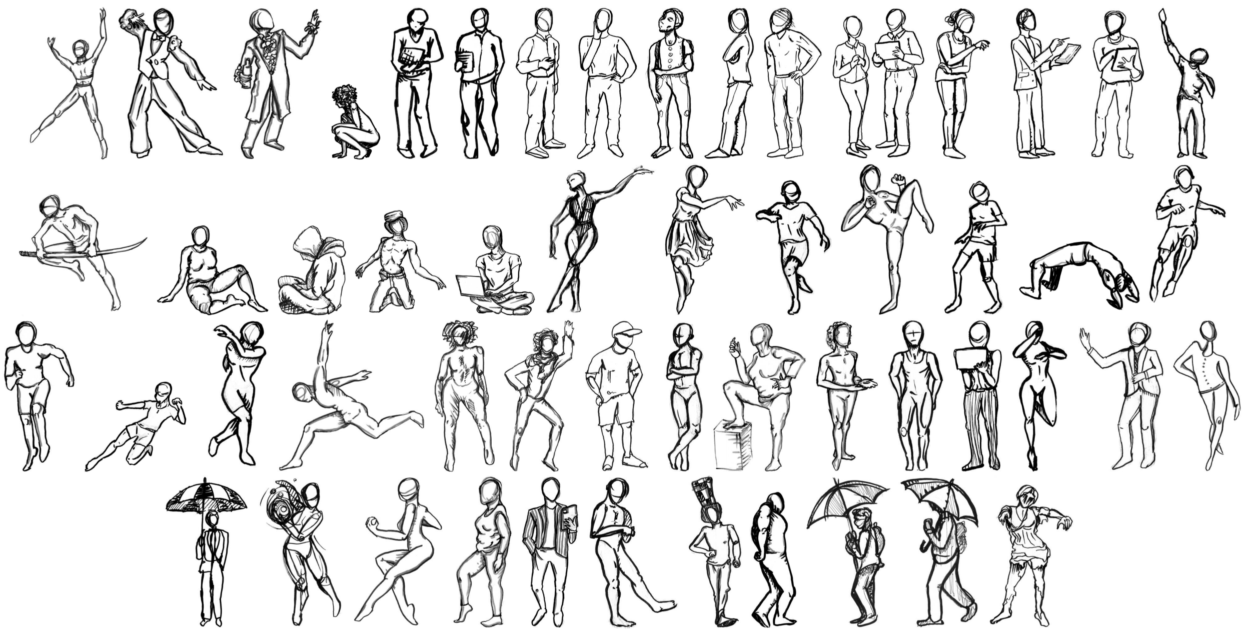


Figure 2: *54 hand-drawn characters that form the basis of our examples and installation.*

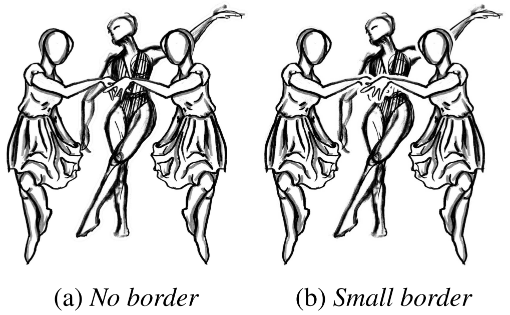
The main mathematical innovation in this installation is our algorithm for randomly laying out characters in perspective while appearing to have a uniform or controlled density. The algorithm also offers several tunable parameters to make a wide variety of layout families; each layout family also offers infinite variation by rerolling the random numbers. We implemented the algorithm in a web app[1] that allows users to set the parameters and immediately see the resulting layout (either in a rectangle or on the curved crease pattern).

In this paper, we present a sequence of algorithms, where each algorithm is a small tweak to the previous algorithm, starting with a very simple random layout and ending with our final algorithm. This progression allows us to describe each idea that we developed, while illustrating its effect with example outputs. The intermediate algorithms also produce pleasing results themselves, so they are of independent interest.

Drawing Aside

Although the main focus of this paper is on the layout algorithm, we start with a brief description of the drawings that we use in all of our examples. We (specifically M. Bass) looked at photographs of people performing various activities, and hand-drew them in an abstract style (in particular, without faces) using black and grey pens on paper. See Figure 2.

Then we scanned these drawings, and Photoshopped them to improve contrast and erase the background to be transparent, so that characters can stack on top of each other. In fact, we left a small white border around each character, to visually separate the layers better. Figure 3 shows the resulting effect.

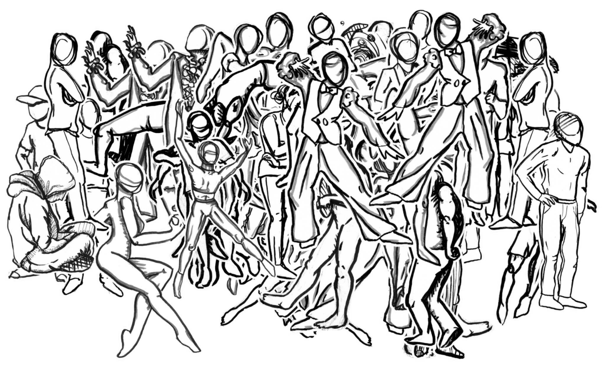


(a) *No border* (b) *Small border*

Figure 3: *A small white border more clearly separates the layers.*

In addition to each character as drawn, we consider their horizontal reflection (through a vertical line); Figure 3 shows an example with the dancer. This trick effectively doubles the number of characters from 54

[1]The web app was written in Civet (a language that compiles to TypeScript/JavaScript) using the SolidJS reactive framework, with SVG for the rendering engine. We plan to open-source a version at https://erikdemaine.org/curved/HangingOut/.


(a) *50 characters at 75% height in random order*

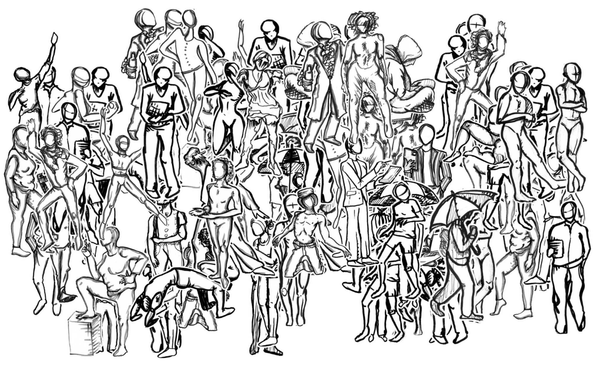

(b) *100 characters at 50% height in random order*

Figure 4: *Examples of completely random layouts, with uniform random stacking order.*

to 108, increasing the visual variety. It also enables interesting interactions between characters, which can be facing either toward or away from each other.

For each character, we define its *natural height ratio* r according to how it should be sized relative to other characters. Most characters all have the same 100% height, while some sitting or otherwise active characters get scaled down to 50%, 60%, 70%, 80%, or 90% height. See Figure 2.

So Random

We start with a simple random layout, which places the characters at uniform random x and y coordinates within a specified rectangle called the *bounding rectangle*. See Figure 4. The main parameters here are how tall the characters are relative to the bounding rectangle ($h_{\max}$), and how many characters to place (n). Figures 4(a) and 4(b) show two different settings: 50 characters at 75% height, and 100 characters at 50% height, respectively. Here 50% or 75% defines the *maximum* height $h_{\max}$ of a character (as a fraction of the height of the bounding rectangle); the actual height $h = h_{\max} \cdot r$ is scaled down by the character's natural height ratio r.

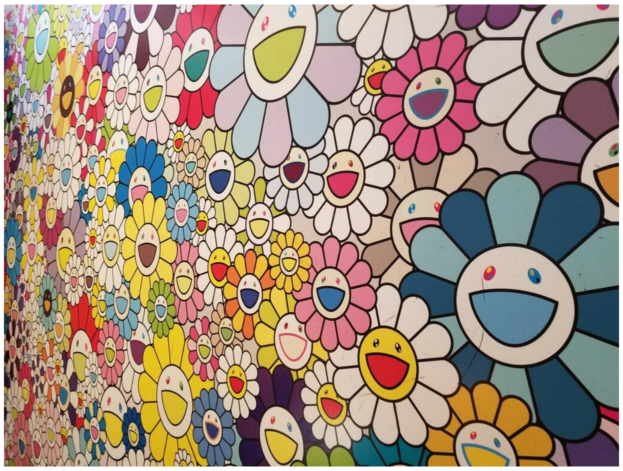
In these diagrams, the stacking order of characters is uniform random. In other words, the layout algorithm is as follows: for each of n times, we choose a uniform random character to place, compute their height $h = h_{\max} \cdot r$, compute their width (according to aspect ratio), choose uniform random x and y coordinates among values that place the character within the bounding rectangle, and then draw the character on top of all previous characters.

These randomly stacked layouts feel similar to Murakami murals of repeating characters, such as his sunflowers and kaikai/kiki characters. See Figure 5.

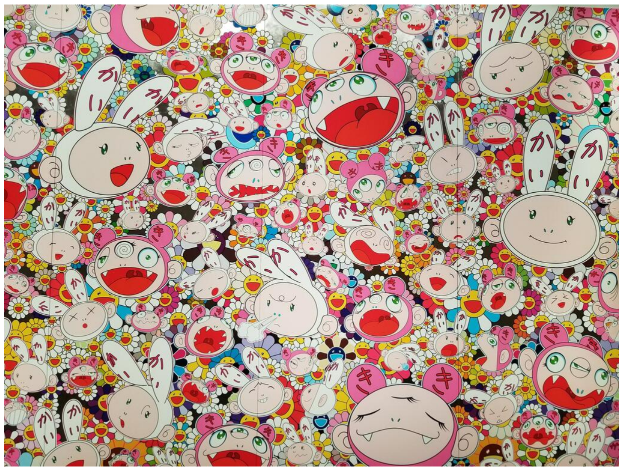
Sorting Out

Our goal was to make the characters seem like they were interacting in a more natural scene. In this setting, characters have a natural depth order, from back (higher y coordinate) to front (lower y coordinate). Figure 6 shows the result of sorting the randomly generated characters in this order, with Figures 6(a) and 6(b) showing the same two parameter choices as Figures 4(a) and 4(b) (but with different random choices).

These layouts essentially correspond to *orthographic projection*, with each character having the same height $h = h_{\max} \cdot r$ in each instance. Decreasing the height parameter $h_{\max}$ essentially corresponds to increasing the height of a camera looking down on the scene.

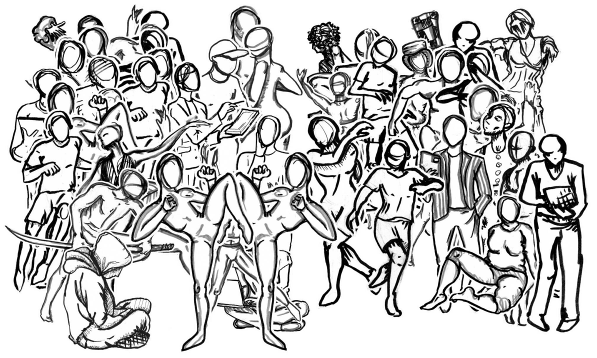


(a) *Printed floor mural accompanying "Kawaii–Vacances: Summer Vacation in the Kingdom of the Golden" by Takashi Murakami (2008).*

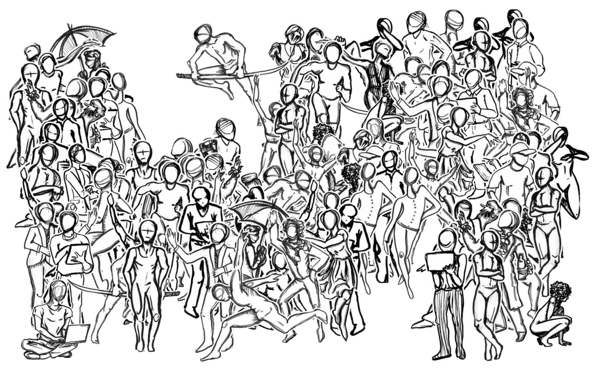


(b) *"Lots, Lots of Kaikai and Kiki" by Takashi Murakami (2009). Acrylic and platinum leaf on 5-panel canvas mounted on aluminum frame.*

Figure 5: *Examples of Murakami's murals of repeating characters, from the exhibit "Takashi Murakami: Lineage of Eccentrics" at the Museum of Fine Arts, Boston (2017). Photographs by E. Demaine.*



(a) *50 characters at 75% height in y-sorted order*



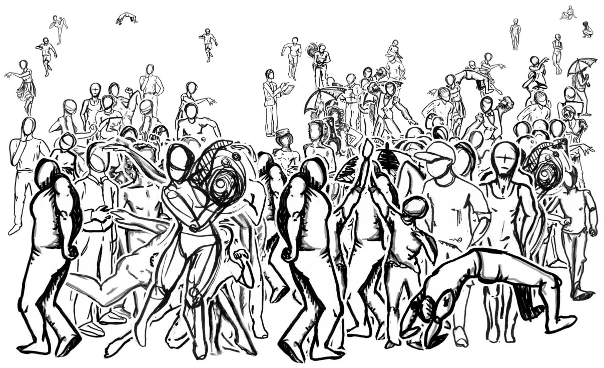
(b) *100 characters at 50% height in y-sorted order*

Figure 6: *Examples of random-coordinate layouts, with stacking order defined by y coordinate. Corresponds to orthographic 3D views. Compare with Figure 4.*

Gaining Perspective

We are more used to seeing scenes in *perspective*. We can achieve this effect by scaling the characters smaller when their y coordinate is larger (farther back), as shown in Figure 7. More precisely, for each character, we choose a random value z between 0 and 1, indicating *closeness* to the camera, and then place the character as follows.

First, we use z to determine the character's scale: we affinely map z to be between two parameters $h_{\min}$ and $h_{\max}$, and use the resulting value $\hat{h} = (1 - z) \cdot h_{\min} + z \cdot h_{\max}$ to determine the character's height (again measured as a fraction of the bounding rectangle's height): $h = \hat{h} \cdot r$. Here $h_{\max}$ effectively controls the height of the camera looking down on the scene: a camera at eye level would have $h_{\max} = 1$, while a higher camera would have $h_{\max} < 1$. Figures 7(a) and 7(b) show two different choices for $h_{\max}$: 65% and 50%, respectively. Parameter $h_{\min}$ prevents characters from being too small; we usually set $h_{\min} = \frac{1}{10} h_{\max}$. The character's scale also affects the character's placement in x coordinate, which is chosen uniformly in the range that keeps the entire character within the bounding rectangle.


(a) *100 characters at 65% maximum height*

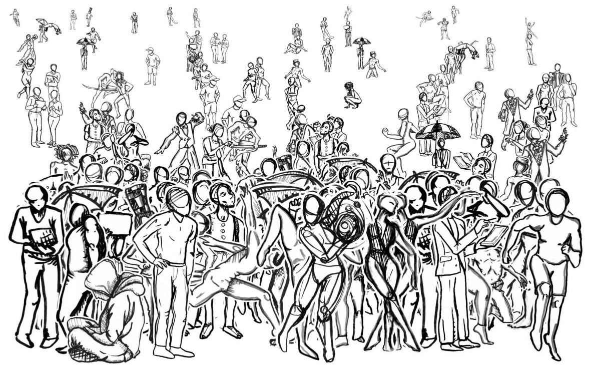
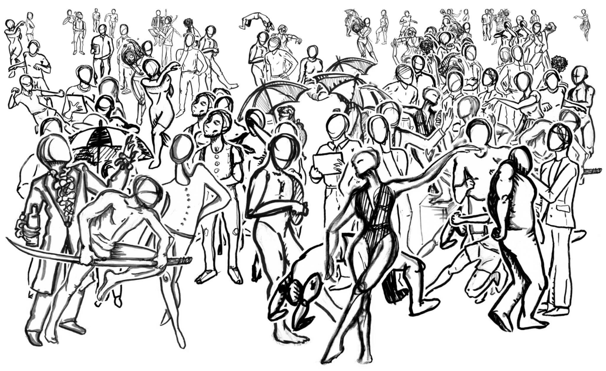

(b) *150 characters at 50% maximum height*

Figure 7: *Examples of perspective layout, with uniform random z values. Background appears sparse.*


(a) *100 characters at 65% maximum height*

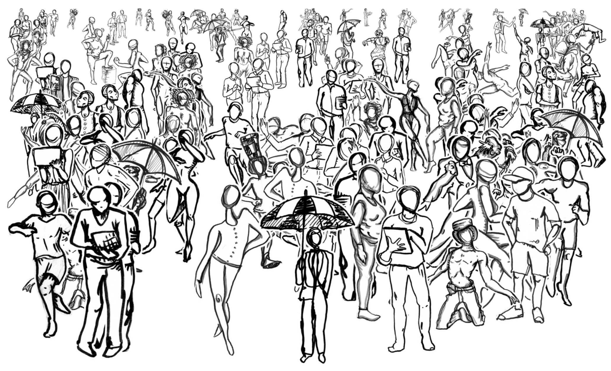

(b) *150 characters at 50% maximum height*

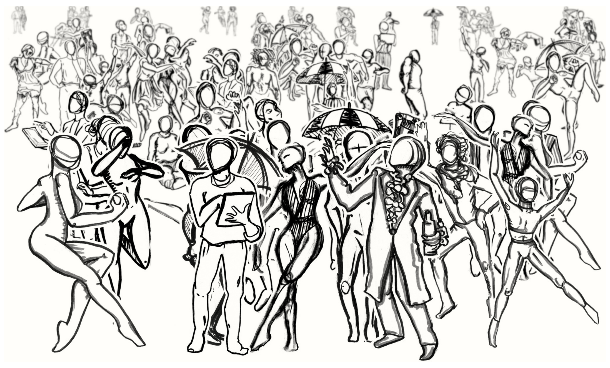
Figure 8: *Examples of perspective layout, with exponentiated random z values. Density appears uniform.*

Second, we use z to determine the character's y coordinate. Consider the range of y coordinates that place the character inside the bounding rectangle, according to the now-fixed scale. Then we select the y coordinate that is a z fraction down from the top of the range. (In fact, the y range is relative to the character's full height $\hat{h}$, not natural height h, so that characters of varying natural height are bottom-aligned.) As before, we use the y coordinate (or z) to determine the stacking order of characters.

Appearing Uniform

The layouts in Figure 7 are unsatisfying because the background feels very sparse, while the foreground feels too crowded. The problem is that we have defined z to be uniformly distributed, but characters in back (with smaller z) are smaller and so do not fill the space as much as characters in front (with larger z). A natural solution is to use a nonuniform distribution for z, i.e., place relatively more characters farther back.

In fact, we found it useful to define two different z parameters: z_1 which determines the character's scale (as described above), and z_2 which determines the character's y coordinate (as described above). After some experimentation, we found good distributions to be $z_1 = \overline{z}^{1.5}$ and $z_2 = \overline{z}^3$ where $\overline{z}$ is a common number chosen uniformly from $[0, 1]$. Figure 8 shows the results, with the same parameters as Figure 7.


(a) *100 characters at 65% maximum height*

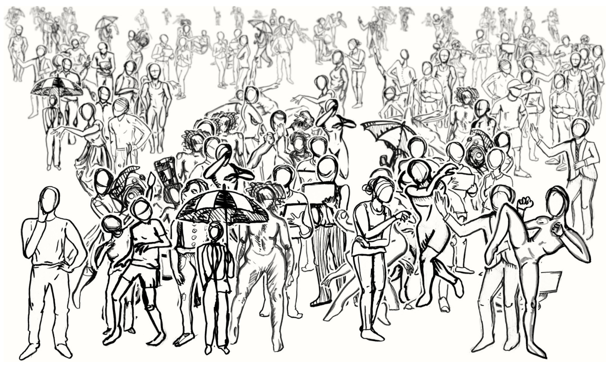

(b) *150 characters at 50% maximum height*

Figure 9: *Examples of perspective layout with blur. Simulates low depth-of-field camera.*

Deep Blur

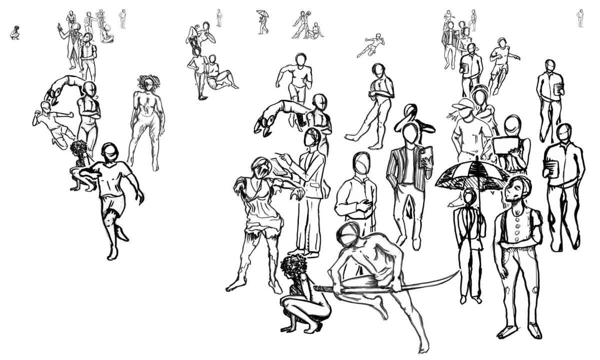
We can make the perspective layout "pop" more by adding a low depth-of-field camera effect, where characters farther in the background get increasingly blurred. After some experimentation, we found that a Gaussian blur with standard deviation $\sigma = \frac{1}{2}(1 - z_1)$ produced visually appealing results, as shown in Figure 9.

Overall, we found the blur effect to look nice on screen, but less so in print on a large-scale sculpture. Thus we ended up not using this effect in the final installation, and omit it from other examples.

(Don't) Be Dense

Now that we have a nice perspective layout, we explore the possible variations for making a variety of different prints (beyond just rerolling the random numbers). We have already illustrated the effect of varying the maximum height $h_{\max}$ of the characters, or effective camera angle. The main remaining parameter at this stage is the number of characters to draw, n, which (together with $h_{\max}$) controls the characters' *density*.

Figure 10 shows two variations on the parameters from Figures 8(b) and 9(b), decreasing and increasing respectively the number of characters by a factor of 3. When selecting these examples, we found ourselves looking at several more random trials in the case of lower density (Figure 10(a)). We suspect that, as the density gets larger, randomness appears more uniform, so the results are more likely to look nice.


(a) *50 characters at 50% maximum height*

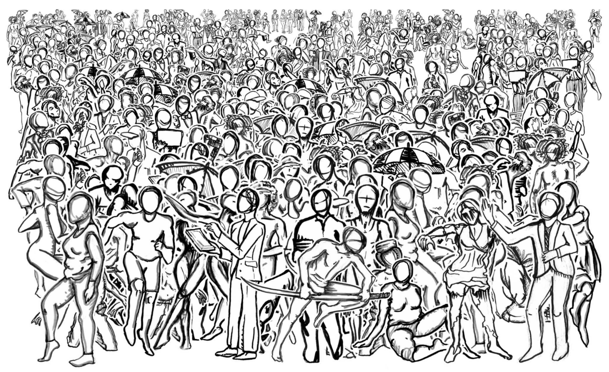
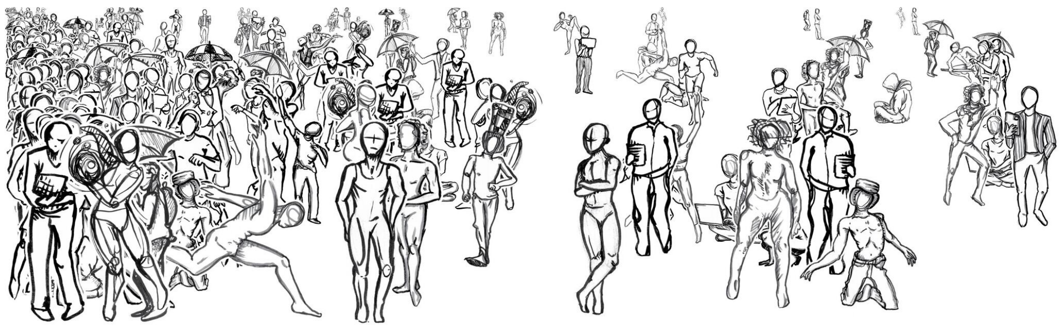
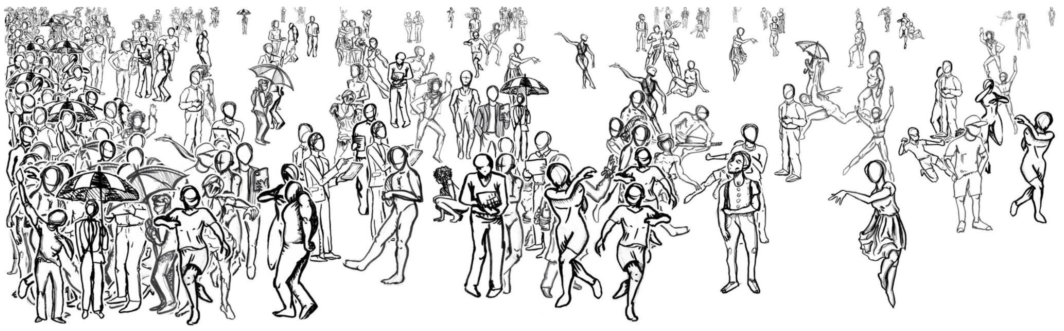

(b) *450 characters at 50% maximum height*

Figure 10: *Examples of varying the density of characters. Compare with Figures 8(b) and 9(b).*



(a) *200 characters at 65% maximum height*



(b) *250 characters at 50% maximum height*

Figure 11: *Examples of left-leaning distributions with $x = \overline{x}^3$. Density is higher on the left.*

Leaning Left

The layouts so far achieve a nice uniform-density look. For variety, we wanted the distribution to vary from one end of the paper to the other. In other words, we wanted the density to be higher at one end of the bounding rectangle and lower at the other end.

Again we found that our friend, exponentiation, was an effective way to modify a uniform distribution into a nonuniform distribution. Specifically, among the range of x coordinates that place the character inside the bounding rectangle, we choose an $x = \overline{x}^\alpha$ fraction from the left (or right), where $\overline{x}$ is a number chosen uniformly from $[0, 1]$ and α is a user-specified parameter. Our final installation used $\alpha = 3$ in some pieces.

Figure 11 illustrates the result. These examples use a much wider bounding rectangle than the previous examples, to give room to visualize the variation in density.

Best Friends

Another flexibility we have is in the random choice of which character to display. Instead of choosing each character uniformly at random, we can bias this distribution toward certain characters. For example, Figure 12 makes the forward-facing "umbrella man" 10 times more likely to be selected, and the two side-facing umbrella carriers each 5 times more likely to be selected — a distribution used in one piece of our final installation. (Our focus on umbrellas comes from "umbrella man" being the first selected drawing, which formed the basis in style for all other drawings.)

Where's Waldo?

The software allows the user to interactively drag characters from their random placement, and to flip each character horizontally *or vertically*. We used the last feature to create an effect like "Where's Waldo"

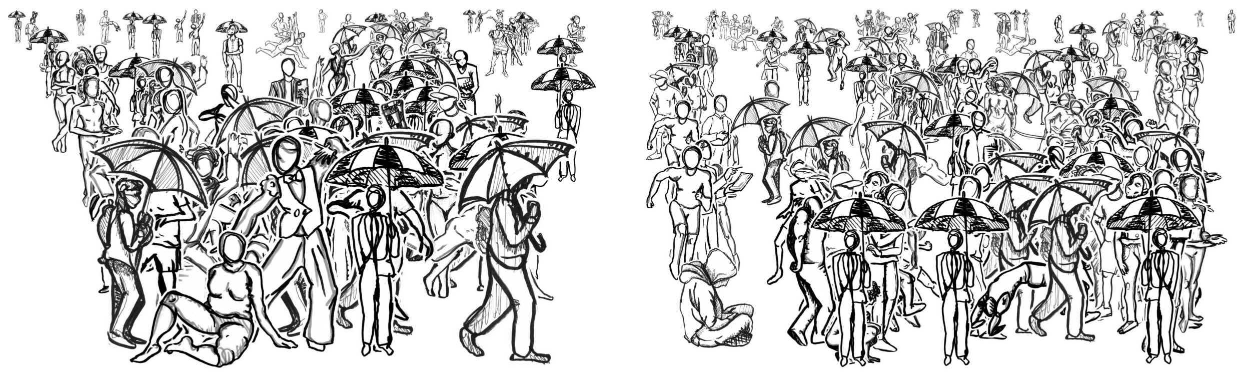

(a) *100 characters at 65% maximum height* (b) *150 characters at 50% maximum height*

Figure 12: *Examples of uneven distribution of character selection. So many umbrellas.*

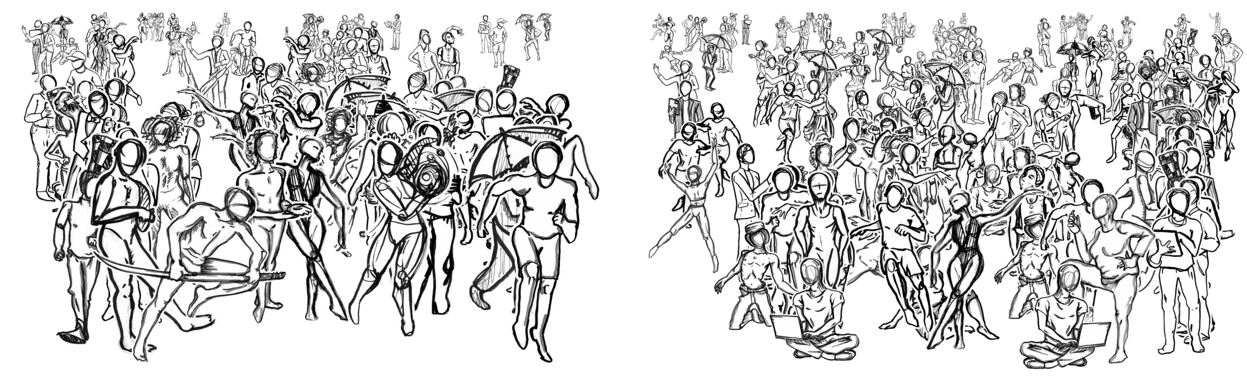

(a) *100 characters at 65% maximum height* (b) *150 characters at 50% maximum height*

Figure 13: *Examples with one upside-down character each. Can you find them?*

("Where's Wally" for those outside Canada and the USA — a book that spent 93 weeks in the New York Times Best Sellers List): we flip exactly one character upside-down in each design, and then the viewer can try to find it. Figure 13 shows two examples.

We also tried other effects, like coloring one character differently, but this was too easy to pick out. Other than drawing a new special character for this purpose, we found that vertical flipping produces results that looked much like a character while still being visually distinctive.

Between the Folds

The final task for our software is to map these layouts onto the crease pattern, shown in Figure 14. Our approach is to construct a rectangular layout for each wiggling *strip* of material between consecutive curves (such as the one highlighted in yellow). The idea is simple: for each character in the layout, convert the x coordinate into the distance along the strip (using angular coordinates within each circle), the y coordinate into the distance across the strip (using radial coordinates), and rotate the character according to the tangent at that point. This conversion would be a Cartesian-to-polar mapping (shown by the cyan grid in Figure 14) if we were using a single circle, but the piecewise-circular crease pattern makes it more complicated.

The first (incorrect) approach we tried is to divide the x range into eight equal segments, and map each range to the corresponding patch of concentric circles (outlined by the purple dashed lines in Figure 14). Specifically, the first segment maps to the first patch; the second and third segments map to the second patch; the fourth and fifth segments map to the third patch; and so on. Figure 15 shows an example result from

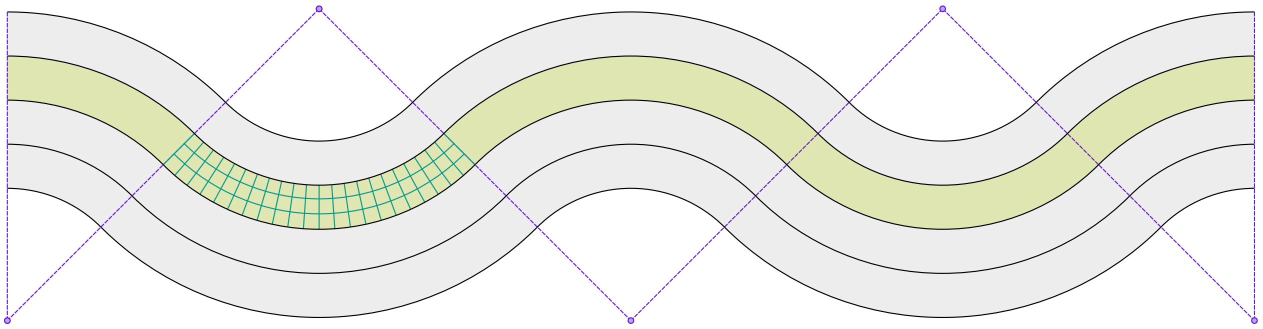


Figure 14: *Crease pattern for each piece in our installation. Purple dots denote circle centers, and purple dashed lines outline concentric circular pieces. Bounding rectangle dimensions: 9.43ft × 2.31ft. Total curved crease and cut length: 31.4ft.*

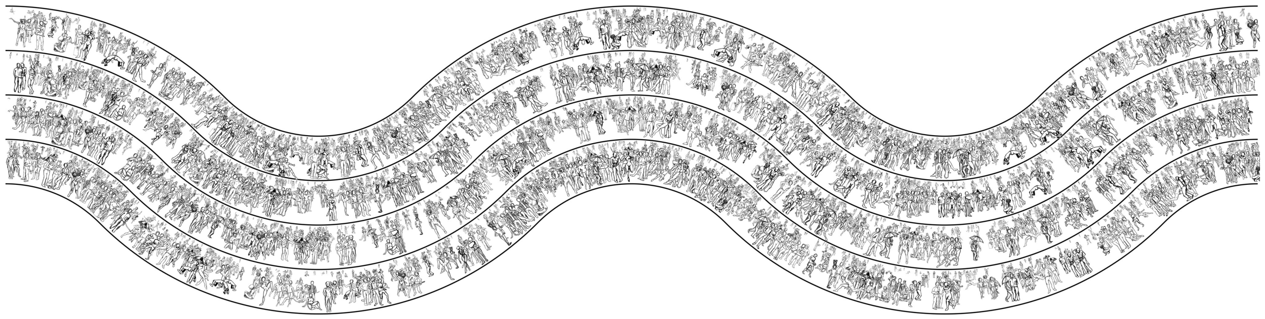


Figure 15: *Treating each circle as an equal portion of the strip, with 1000 characters per strip (at 75% maximum height). Characters are denser in areas of higher curvature (closer to circle centers).*

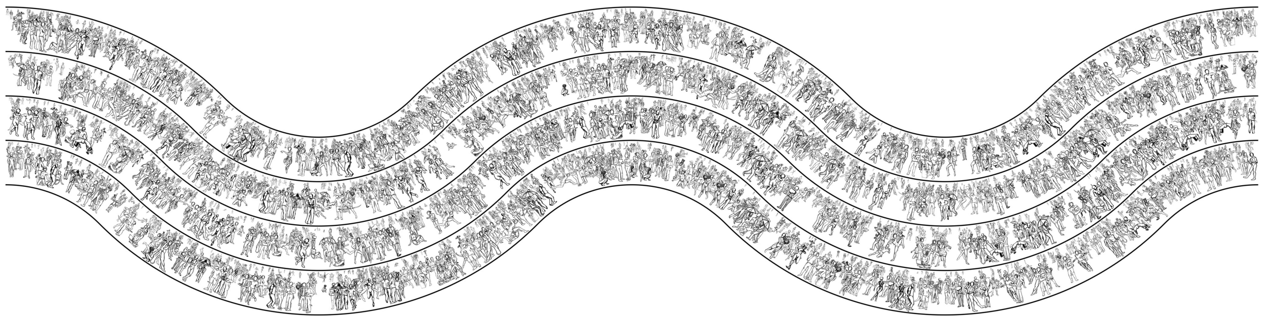


Figure 16: *Correctly distributed layout with 1000 characters per strip (at 75% maximum height). Compare with Figure 15.*

this approach, where the distribution of characters is subtly nonuniform (most obviously in the topmost and bottommost strips). The problem is that the same angular range (and thus x coordinates) has different lengths on different circles, increasing with the radius. Thus we end up with characters more densely packed in areas closer to circle centers, while each strip alternates between close and far and thus between dense and sparse.

The correct (uniform) approach is to distribute the x range to different circles according to the pair of radii given at the assigned y coordinate. The messy part is that there are two radii involved, r_1 for the circles with centers at the bottom and r_2 for the circles with centers at the top. Then we allocate x coordinates as follows: the initial interval of length $r_1 \cdot 45°$ maps to the first circle; the next interval of length $r_2 \cdot 90°$ maps to the second circle; the next interval of length $r_1 \cdot 90°$ maps to the third circle; and so on. Figure 16 shows an example of the resulting distribution, with the same parameters as Figure 15; the result is noticeably more uniform at the areas of high curvature (nearest the circle centers).

Hanging Out

The final installation, illustrated in Figure 1, follows essentially the algorithm described above. The only difference is that we vertically flipped the character layouts in two of the four strips, so that the characters had similar orientations after folding along the creases. We also flipped one character in each strip upside-down, so each piece of paper has four "Where's Waldo?"-style puzzles.

We printed each sheet using a Canon imagePROGRAF Pro-4100 Printer fed with 3.5ft-wide rolls of Canon Matte Paper.[2] Indeed, the near-infinite length of paper rolls is one of the main inspirations for this project. We ended up choosing the 9.43ft $\times$ 2.31ft "two-wave" crease pattern of Figure 14 because it offered enough length to do interesting folding, while not being so long that its weight would be difficult to support.

Paper rolls do not like to be fed back into a printer upside down (and most paper has a preferred side to print on), yet we wanted our imagery to be double sided, with each character being visible from both sides. We ended up printing each sheet twice (one in reflection), and stacking the two folded sheets on top of each other, so that the front and back imagery matched up nearly exactly. Surprisingly, the adjacent curved folds held together the two sheets rather well without any adhesive. In a few cases, the hanging weight was enough for the two sheets to separate, which we resolved with a few staples.[3]

The next step was hanging the pieces, which required a way to connect monofilament (fishing line) to the paper without causing the paper to rip. We found that one binder clip on either end of the folded sheets was enough to hang the paper, presumably thanks to a combination of the curved creases and the two layers of material. In some cases, we added a third clip, so that we could further control the folded form. Overall, we found that hanging enabled the expression of several different shapes, though it was initially difficult to predict how the forms would hang until we did so. (Luckily, it was easy to rehang until we liked the results.)

Summary and Conclusions

We plan to release an open-source version of our code so that users can upload their own set of images and make their own orthographic or perspective layouts. Watch the installation website for updates.[4]

We plan to experiment with workshops for children, where they draw their own characters (monsters, etc.), we scan them, the software immediately generates pleasing random layouts, and we can print them out.

Acknowledgements

Many thanks go to the Fitchburg Art Museum for inviting and housing this project. In particular, the curator Lauren Szumita asked whether we could do any larger pieces than our usual $\approx$ 1ft scale, which we misinterpreted as a *requirement* to go bigger, inspiring this project. Jesse Kenas-Collins constructed the wooden trellis that we hung our pieces from, and played a major role in installation.

Many thanks also go to The Infrastructure Group at MIT CSAIL, in particular Jason Dorfman, for getting the Canon poster printer working in time to make this installation possible.

[2]Funny construction story: Originally, we planned to print on an Epson SureColor T7270 Single Roll Edition Printer, but it stopped working (no ink came through the printhead) exactly when we needed to start printing the final sheets. Luckily, the new Canon printer arrived shortly thereafter and got installed just in time to save the day.

[3]Another funny construction story: Originally we thought that it would be best to staple each pair of sheets all along their boundary, so we initially added staples every few inches. Later we realized that slight mismatches led to the paper buckling between staples, which was unattractive. So we ended up *removing* all of the staples, and then adding a few where necessary after hanging. Stapling and unstapling is an effective way to waste a couple of hours.

[4]https://erikdemaine.org/curved/HangingOut/