Automating the Intentional Encoding of Human-Designable Markers

Joshua D. A. Jung, Rahul N. Iyer, Daniel Vogel
Cheriton School of Computer Science
Waterloo, Canada
{j35jung,r3iyer,dvogel}@uwaterloo.ca

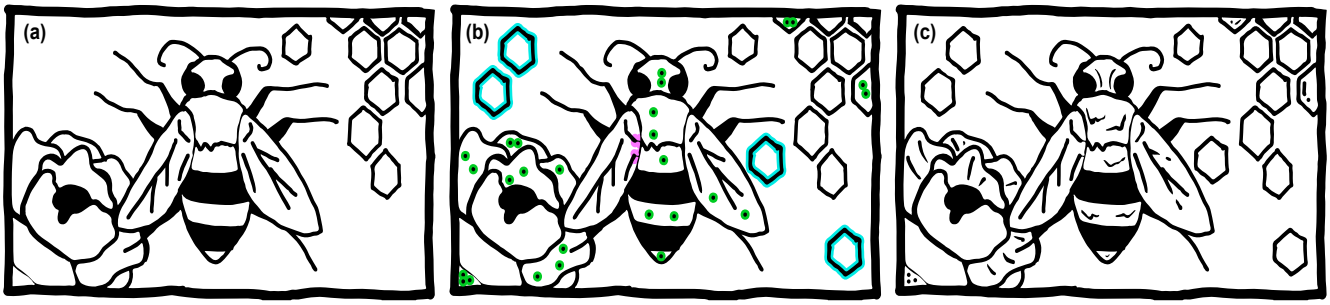

Figure 1: Drawing stages incorporating automated methods for a marker encoding 'o4p8c7u' (tinyurl.com/o4p8c7u redirects to the 'Honey Bee' Wikipedia page): (a) The artist's initial drawing; (b) The drawing modified by the algorithm with added "dot blobs" (highlighted in green), repeated shapes (highlighted in cyan), and connecting lines (highlighted in magenta); (c) The final drawing after artist touch-ups.

ABSTRACT

Recent work established that it is possible for human artists to encode information into hand-drawn markers, but it is difficult to do when simultaneously maintaining aesthetic quality. We present two methods for relieving the mental burden associated with encoding, while allowing an artist to draw as freely as possible. A 'Helper Overlay' guides the artist with real-time feedback indicating where visual features should be added or removed, and an 'Autocomplete Tool' directly adds necessary features to the drawing for the artist to touch up. Both methods are enabled by a two-part algorithm that uses a tree-search for finding 'major' changes and a dynamic programming method for finding the minimum number of 'minor' changes. A 24-person study demonstrates that a majority of participants prefer both tools over previous methods of manual encoding, with the Helper Overlay being the more popular of the two.

KEYWORDS

fiducial marker, designable marker, encoding, user study

1 INTRODUCTION

The ubiquity of the modern smartphone camera has allowed visual markers, like QR codes, to transmit data through a diverse array of non-electronic media. There are textbooks that link to demonstrations, museum plaques that queue up recordings, and, perhaps most commonly, advertisements that link to product websites. Despite this utility, QR codes and similar machine-generated markers miss an opportunity for meaningful interaction: to a human observer, they all look the same. Further, the rigidity of their design precludes them from maintaining a consistent aesthetic with their surroundings in cases where it is desirable to do so.

Human-designable visual markers introduced by Costanza and Huang [7], and extended by Preston et al. [18], address these issues by allowing a human artist to exercise great control over the design of a marker. To a human observer, a marker of this type appears to be a simple drawing, but it also contains data for a machine to scan. However, such markers present new challenges. In particular, it is often desirable to encode a predetermined value, as is the case when encoding a URL. Because every brushstroke contributes to both the marker's encoding and its aesthetic, the artist must always

CHI 2019, May 4–9, 2019, Glasgow, Scotland Uk

ACM ISBN 978-1-4503-5970-2/19/05…$15.00
https://doi.org/10.1145/3290605.3300417

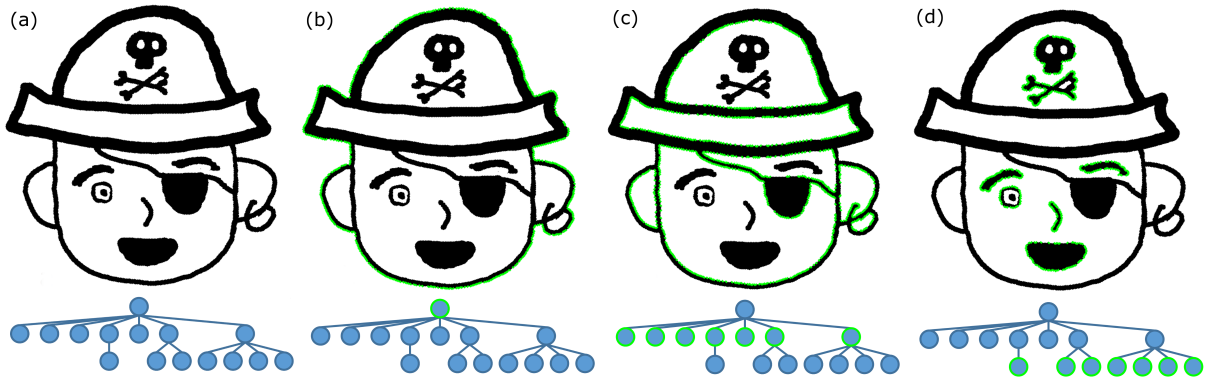

Figure 2: Highlighted contours and their positions in the region adjacency tree. (a) Original image. (b) Root contour. (c) Part contours. (d) Blob contours.

keep both in mind. This *intentional encoding problem* was introduced by Jung and Vogel [14], who showed it to be solvable by human artists, but that doing so can be artistically restrictive and mentally taxing.

We introduce two methods that incorporate varying degrees of automation into this process to relieve some of the burden on the artist. The first, called the 'Helper Overlay', separates the encoding into manageable pieces and gives instruction on roughly where to add new elements. The other, called 'Autocomplete', allows the artist to draw freely (Figure 1a), then press a button to automatically alter the marker to achieve the desired encoding. Alterations include adding 'dots', repeating shapes, and connecting shapes together (Figure 1b). Afterwards, the artist may continue to touch up the new elements in ways that do not change the encoding (Figure 1c). Introducing automation comes at some cost to user agency, a trade-off that has been studied in other mixed initiative systems [11, 16]. We hypothesized that most users would be willing to relinquish control over design elements that principally interact with the encoding and have minimal impact on artistic freedom. This is consistent with the work of Alan et al. [2], who found that users tend to be comfortable with automation as long as they still feel in control.

To test this hypothesis, we conducted a 24-participant study to assess the usability of our methods relative to the 'Manual' method of encoding used by Jung and Vogel [14]. We find that most users do indeed prefer these new methods, with the Helper Overlay placing first, Autocomplete second, and Manual a distant third.

The primary contributions of this paper are:

1. An algorithm for finding the minimal set of changes to make a drawing match a specific encoding;
2. A pair of interactive methods that leverage this algorithm to automate much of the encoding process;
3. A user study with amateur designers that evaluates how these methods compare with manual encoding.

2 BACKGROUND AND RELATED WORK

If we view the current encoding of some partially complete marker as an initial bit string, and the desired encoding as a final bit string, then finding the minimal number of required alterations may be seen as a variant of the *Approximate String Matching* problem [17]. However, the action of inserting a 'divider' into the target encoding (described in Section 4) is not typical for this problem, and is notably different from the addition, subtraction, or substitution actions that are often associated with the edit distance.

One could consider our problem to be a variant of string partitioning, or related problems where a linear structure must be divided into smaller pieces, such as rod-cutting [6]. Our situation is unusual in that the insertion of a divider in one location may preclude insertion in others. Additionally, we must consider factors particular to drawing, such as the finite space inside a marker means only a limited number of shapes may be drawn in one area.

Computer-Generated Artwork

A central theme of this work is the preservation of aesthetic quality in artwork that is guided by, or in the case of Autocomplete, partially generated by a machine. This is a goal shared by those in the field of computer-generated artwork (CG-artwork), as surveyed by Boden and Edmonds [4]. Ha and Eck [10] provide an example of this work that is similar to our own. They feed a recurrent neural network (RNN) with hand-drawn images of a given topic (e.g. 'pig') and train it to reproduce similar drawings. This ability also allows it to complete drawings started by a human user, which is similar to the function of our Autocomplete tool. However, it is not clear how a neural network could be applied to the problem of intentional encoding. The appearance of a pig may be approximated, but an encoding must be exactly correct or it is useless. We revisit this discussion in Section 7.

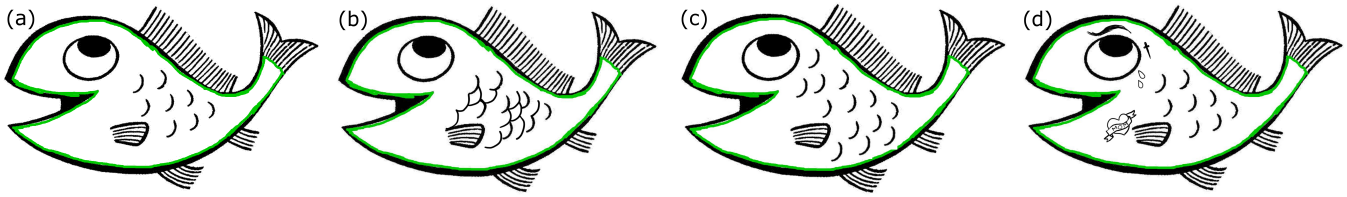

Figure 3: Fish image used in the blob placement survey. (a) Original image. (b) Reduced blob number by connecting shapes. (c) Increased blob number by duplication. (d) Increased blob number by addition of new details.

Topological Visual Markers

Barcodes and QR codes make use of *geometric* features in which the size and shape of black regions is important to the overall encoding [12, 20]. Though information dense, such encoding schemes set tight constraints on marker design, so they do not lend themselves to the domain of human-designability. On the other hand, schemes that employ *topological* features are unchanged by continuous deformations, which allows for the possibility of markers that look very different, but represent the same encoding.

One such feature, the *region adjacency tree (RAT)*, is determined by finding the contours of an image. These become nodes in the tree, with a contour that is nested inside another contour in the image becoming a child node of that larger contour in the tree (see Figure2). Johnston and Clark's Vision Positioning System [13] and reacTIVision [3] both use the RAT as a marker identifier. This is done to make use of desirable recognition features like rotation and skew invariance, not to enable human-designability.

Human-Designable Markers

The potential of the region adjacency tree to be leveraged for aesthetic purposes was realised in the d-touch recognition system [7–9]. It employs an encoding scheme that considers the top three levels of the tree, comprised of:[1]

1. A single outer *root* contour (Figure 2b);
2. Some number of *part* contours within the root that enclose separate white areas (Figure 2c);
3. Some number of *blob* contours within the parts that enclose free-floating black shapes (Figure 2d).

Contours deeper in the RAT are ignored. To find the encoding of a tree in the d-touch system, every part is assigned a value equal to the number of blobs it contains (i.e. its children in the tree). These values are then combined by arranging them from smallest to largest. For example, the pirate marker in Figure 2 produces the encoding (0,0,0,0,1,2,4) as a tuple, or 0000110100 in binary, which we obtain by converting all values in the tuple to binary. Every tuple has exactly one corresponding binary value, but the converse is not true: (0,0,0,0,52) also produces 0000110100. In general, there may be many valid markers that represent the same binary encoding. We may note the difference by writing the binary values like 0|0|0|0|1|10|100 and 0|0|0|0|110100, where each '|' is a divider symbol denoting the beginning of a new part.

A drawback of this system is the difficulty associated with representing a binary value like 10000000. Because the values in the tuple must be in non-decreasing order, (1,0,0,0,0,0,0,0) and similar tuples are not valid[2]. Instead, (128) must be used. That is, if this were the desired encoding, an artist would have to draw a marker with only 1 part containing 128 blobs. This property renders the original d-touch system impractical for intentional encoding, in general. Preston et al. [18] alleviate this issue by redefining how the part values are ordered. Instead of sorting by value, parts are arranged from smallest to largest by area occupied in the marker. Though this not a topological feature, it decouples a part's value from its position in the encoding, making tuples like (1,0,0,...) legal.

Intentional Encoding

Jung and Vogel [14] introduce the intentional encoding problem, which requires the artist to encode a predetermined binary value. In a lab study, participants were asked to perform this task using several different encoding schemes. The scheme defined above, referred to as Number/Area (i.e. count the **number** of blobs and arrange parts by **area**) was most preferred by participants. It will therefore be the encoding scheme used throughout the rest of this paper.

Though the study found it was possible for participants to complete markers encoding as many as 20 bits (approximately half of a URL shortened via a service like tinyURL [1]), doing so was extremely mentally taxing. Much of this difficulty stemmed from the method of splitting up a binary value into parts, which required participants to place dividers (|) manually. This not only presented them with a time-consuming puzzle, but also allowed the possibility of suboptimal solutions requiring more work for them to complete. In our study, we refer to this as the 'Manual' condition, and compare it with two new methods that automatically perform the task of divider assignment for the user.

[1] We make use of the naming convention defined in [14].

[2] Also note that binary values like 00 or 01 cannot be represented, as those would be indistinguishable from 0 and 1. In order for a set of dividers to be valid, all 0s must be alone in a part or behind a leading 1.

3 BLOB PLACEMENT SURVEY

In order to ascertain how an artist might approach the task of adding or removing blobs from a marker, we conducted a survey of 5 trained artists. This was completed on paper or through a commercial digital image editor. The survey consisted of:

- A description of blobs and the ways in which their number can be manipulated;
- Several colouring book pages, on which the artists were either asked to increase or decrease the total number of blobs by a set amount (Figure 3);
- A set of 5 long-form questions asking about the experience and any strategies used.

From the results of this survey, we compiled key qualitative observations that guided the design of our Helper Overlay and Autocomplete tools.

There is no consensus about whether it is easier to add blobs or remove them. One question asked the artists which task was easier, and how many instances of that task would be equal in difficulty to one instance of the harder task. The decision was split 3-2 in favour of removal, with one artist claiming that it was 3 times easier than addition. However, the two artists in the minority both claimed it was 3 times harder. Evidently, it is an issue of personal preference, which we incorporated into the user interface design discussed in Section 5.

Preservation of detail is important. A majority of the artists never chose to outright erase blobs, but instead connect them to the edge of the part or to each other (as in Figure 3b). When asked why, several artists cited a desire to preserve detail. In particular, one artist described wanting to "add more to the image and possibly describe the subject better, without losing detail". With this finding, we designed the Autocomplete Tool such that it uses deletion only as a last resort, as described in Section 4.

When there are many similar objects, duplicating them is a viable strategy for adding blobs. Some of the artists took this approach for images with 'texture' blobs, like the fish scales in Figure 3c. Generating new blobs that make sense in an image is difficult for a machine, but copying existing blobs is relatively inexpensive. The Autocomplete Tool therefore incorporates blob duplication.

Artists create things that are difficult or impossible for a rule-based system to anticipate. Even with the foreknowledge that human creativity is difficult to capture and recreate, we were consistently surprised by the imaginative ways in which blobs were added or removed (e.g. adding tattoos to the fish in Figure 3d). This observation is ultimately the reason that the Helper Overlay was created to guide, rather than automatically fix.

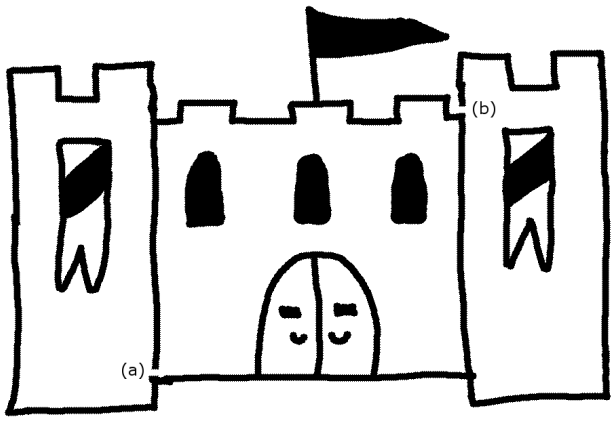

Figure 4: A marker with (a) a small cut between two parts and (b) a small cut between a part and the root. Without cuts, it encodes (2,2,1,1,3). With cuts as shown, it encodes (2,2,4).

4 ALGORITHM

Given a desired binary encoding and an existing image with some number of parts and blobs, we define a 'solution' to be a set of alterations that may be made to the image in order to achieve the desired encoding. These alterations may fall into one of three categories:

1. Increasing the number of blobs in a part;
2. Decreasing the number of blobs in a part;
3. Cutting a part by erasing a small section of its boundary.

Increasing or decreasing the number of blobs can be accomplished directly by adding or deleting a blob, but may also be achieved by other means. Existing blobs may be split in half to increase their number, or they may be joined together or to the edge of a part in order to decrease it. Summing the total number of required blob changes (both increases and decreases), gives a basic cost function that varies with the choice of dividers in the binary encoding. For example, if a partially drawn marker currently has an encoding of (1, 3, 0, 4) and an encoding of 1101110 is desired, we could choose $1|101|1|10 \equiv (1,5,1,2)$. To make the marker match this solution, we would need to add 2 blobs to the second part, add 1 blob to the third part, and remove 2 blobs from the fourth part, for a total cost of $0+2+1+2 = 5$. Alternatively, we could choose $1|10|1|110 \equiv (1,2,1,6)$, with a cost of $0 + 1 + 1 + 2 = 4$. By this cost metric, the latter solution is superior to the former. We describe a real-time algorithm for finding optimal solutions in the 'Finding Optimal Divider Positions' section below.

If a small cut is made on the outer boundary of a marker, a part may be joined to the root, removing it and its contents from the marker entirely. If the boundary between two adjacent parts is cut, those parts are merged. Both possibilities have the effect of reducing the total number of parts by one in an inconspicuous manner, as depicted in Figure 4. The 'Cutting Parts' section below discusses a method for finding

potential cut locations and using them to find even better divider solutions. Note that we do not consider the possibility of automatically finding places to add parts because doing so (for example, by splitting an existing part with a dividing line) is considerably more difficult without sacrificing aesthetics. It is therefore easier to find a good solution for a marker that has too many parts as opposed to too few.

Once a complete solution is found, it is presented to the user through either the Helper Overlay or Autocomplete method. These are described in detail in Section 5.

Finding Optimal Divider Positions

In general, it is not computationally feasible to look for an optimal solution among all possible choices of dividers, even if we require that the number of divisions match the number of parts in the partially complete marker. A naïve solution runs in $O\binom{n-1}{d}$, where n is the length of the desired binary string and d is the number of dividers to be placed (i.e. the number of parts - 1). If we are interested in encodings for which $n \approx 40$ (the approximate size of a shortened URL) and d can take any value between 1 and $n-1$, then the number of operations may be as high as 100 billion. This is untenable for a real-time direct manipulation user interface.

Instead, we present a dynamic programming algorithm, summarised by Algorithm 1, that runs in $O(n^2 d)$ time. This algorithm, called *FindDividers*, operates on the principles that we may, without loss of generality, place dividers one at a time from left to right in the binary encoding, and that we need only know the position of the most recently placed divider and the total cost of partial solutions.

We define a matrix, *costs*, whose first index is the number of dividers placed (minus 1) and whose second index is the position of the most recently placed divider. Values stored in *costs* are determined by some *Cost* function, which takes as parameters the current number of blobs in a part and a target number of blobs. This function may be a simple count of the number of blob additions or subtractions required, but also permits various modifications. For example, we may wish to increase the cost of adding blobs to smaller parts, or to weight blob removals as being costlier than additions, or vice versa. This and other cost function multipliers are discussed further in Section 5. Note that the choice of cost function does not impact the correctness of the final encoding (which always exactly matches the target), but it may significantly influence which dividers are chosen, and consequently affect the set of actions needed to reach the target.

The first **for** block of Algorithm 1 (line 6) initializes the first column of *costs* with all possible locations for the first divider. Note that *BinVal(startPos, endPos)* functions like substring for our desired encoding. That is, it gives the numeric value for digits positioned in the range $[startPos, endPos)$.

Algorithm 1 FindDividers

```
1:  parts ← [number of blobs in each part]
2:  n ← length of binary string − 1
3:  d ← number of dividers (parts − 1)
4:  costs[d][n] ← [[∞, ∞, ...], [∞, ∞, ...], ...]
5:
6:  for j ← 0 to d − 1 do                ▷ Initialize first column of costs
7:      bitVal ← BinVal(0, i + 1)
8:      if bitVal is valid then
9:          costs[0][j] ← Cost(parts[i], bitVal)
10:
11: for i ← 0 to d − 1 do                          ▷ Fill rest of costs
12:     for j ← 0 to n − 1 do
13:         cand[j] ← [∞, ∞, ...]
14:         for k ← 0 to j − 1 do
15:             bitVal ← BinVal(k + 1, j + 1)
16:             if bitVal is valid then
17:                 cand[k] ← costs[i − 1][k]+
18:                               Cost(parts[i], bitVal)
19:                 if i = d − 1 then          ▷ Last divider, 2 parts
20:                     cand[k] ← cand[k]+
21:                                   Cost(parts[i + 1],
22:                                        BinVal(j + 1, n + 1))
23:         costs[i][j] = min(cand)
24: return min(costs[d − 1])
```

The second **for** block (line 11) fills in the rest of the table by adding columns one at a time, from left to right. For any given table entry, there are $O(n)$ candidate values (stored in *cand*), which arise from previous partial solutions with different final divider positions. To each of these, the cost associated with introducing a new divider is added, and then the candidate with minimum cost is entered into the *costs* matrix. When the table has been completed, the set of dividers associated with the minimum cost entry of the last column is returned as the optimal solution.

Cutting Parts

Possible cut locations are determined by analysing the geometry of the partially complete marker. The number of points on every part contour are reduced to a polygon approximation via OpenCV's *approxPolyDP* function [5]. For every pair of parts, we then locate the two points that are closest together. These points are each on two line segments that form part of their respective polygon approximations, and among these line segments, we find the closest points between the two parts. These points are not guaranteed to be the closest points on the full part contours, but we only require an approximation that is visually similar enough to go unnoticed. If all points on the line between the selected points are black pixels, then we have found a valid cut location. White pixels

indicate that a third, unrelated part would be breached, so the cut is discarded if they are found[3].

Once all possible cut locations have been determined, we may use this information to conduct a tree-search. The root node of this tree is the current encoding of the partially complete marker, and its children are the encodings that may be arrived at by performing one of the possible cuts. Their children may be found by performing one additional cut, and so on. As it would be too time consuming to actually manipulate the image for every node expanded, we instead keep track of its RAT, accounting for changes in part ordering and blob numbers. A cost is assigned to each node using the *FindDividers* algorithm, and the cost of performing a cut is assumed to be zero. (We revisit this decision in Section 7.) Using Dijkstra's Algorithm, we search for the node with lowest cost, stopping when either all nodes have been expanded, or a time limit has expired. The selected node represents a complete solution: a set of cuts to perform, and the divider positions that minimize cost for the marker after cutting.

Autocompletion

If using the Helper Overlay, this is where the machine's work is finished; it need only display the solution, as described in Section 5. Autocomplete, however, must determine how it will effect the solution. Performing a cut is straightforward, requiring only that a white line be drawn between the two points specified. However, there are several methods for both adding and removing blobs with varying levels of aesthetic impact. For each, we define a procedure that attempts aesthetically 'better' methods before resorting to those that are more reliable, but aesthetically 'worse', where a method's value is determined by the survey described in Section 3.

To increase the number of blobs in a part, the tool first tries to find groups of similar blobs and will duplicate a blob from the largest group if one can be found. To detect if two blobs are similar, we use OpenCV's Template Matching algorithm (*matchTemplate*) that calculates the normalized correlation coefficient similarity between the two blob images (cropped from the main image) [5]. This is calculated for all pairwise combination of blobs inside a part. Any pair of blobs that have a similarity coefficient of more than 0.5 are counted as similar and will be considered for duplication. At the end of this process, the blob with the highest number of similarity matches is duplicated. Candidate locations for the new blob are randomly drawn from a uniform distribution over pixel locations confined by the boundary of the part. A candidate is rejected if any pixels of the placed blob would be located outside or too close to the part boundary, or overlapping with or too close to other blobs. If a valid position can be found within 200 attempts, the duplicated blob is drawn in that location. If not, or if no blobs are similar enough to warrant duplication, this process is instead repeated for a small circular blob (a 'dot'), for which a valid location is easier to find. If one still cannot be found, then Autocomplete will leave the part unfinished and fail to achieve the desired encoding. This most often occurs when the artist has drawn very few parts, which must then contain an unreasonably large number of blobs. (For example, if the artist draws a single part for a 20-bit encoding, it would have to contain upwards of 500000 blobs.) When this occurs, the artist must alter the part structure of their marker and activate Autocomplete again. Figure 1b contains examples of both duplicated blobs (cyan) and dots (green).

To reduce the number of blobs in a part, the Autocomplete Tool begins by trying to join blobs together or to the boundary of the part. For every pair of blobs, the minimum distance line to connect them is found, and similarly, minimum distance lines are found between blobs and the boundary. These candidate lines are then selected in increasing order of length and checked for intersections with any other shapes. Only if it cuts through empty space can a line be accepted and drawn. If no suitable lines can be found, then, as a last resort, a blob will instead be deleted to reduce their number, with smaller blobs being deleted before larger ones.

5 USER INTERFACE

We use a version of the open-source drawing application developed by Jung and Vogel[4] [14] (Figure 5), modified to include the Helper Overlay and Autocomplete Tool[5].

Manual

The original version of this application provides all of the functionality required for the Manual experiment condition, in which participants were required to click on a visual representation of the target binary encoding (Figure 5b) in order to find a valid set of dividers. This condition offers the most direct control over the encoding of data, at the cost of the increased cognitive load associated with doing so. Participants were also given access to the Label and Order tools (Figure 5f), which show the encoded values and relative sizes of all parts in the marker. For a full description of these tools, as well as the target encoding block, we defer to [14].

Helper Overlay

The Helper Overlay method takes control of the target encoding panel, focusing the user's attention on the canvas area.

[3]It is possible that some other pair of points, which are not the closest, may provide a valid cut location. However, we do not continue searching. In general, our philosophy for finding cut locations is that false positives are far worse than false negatives. That is, missing an opportunity for a cut is preferable to executing a cut that has unintended consequences.

[4]github.com/jdajung/sketchncode

[5]github.com/jdajung/autosketch

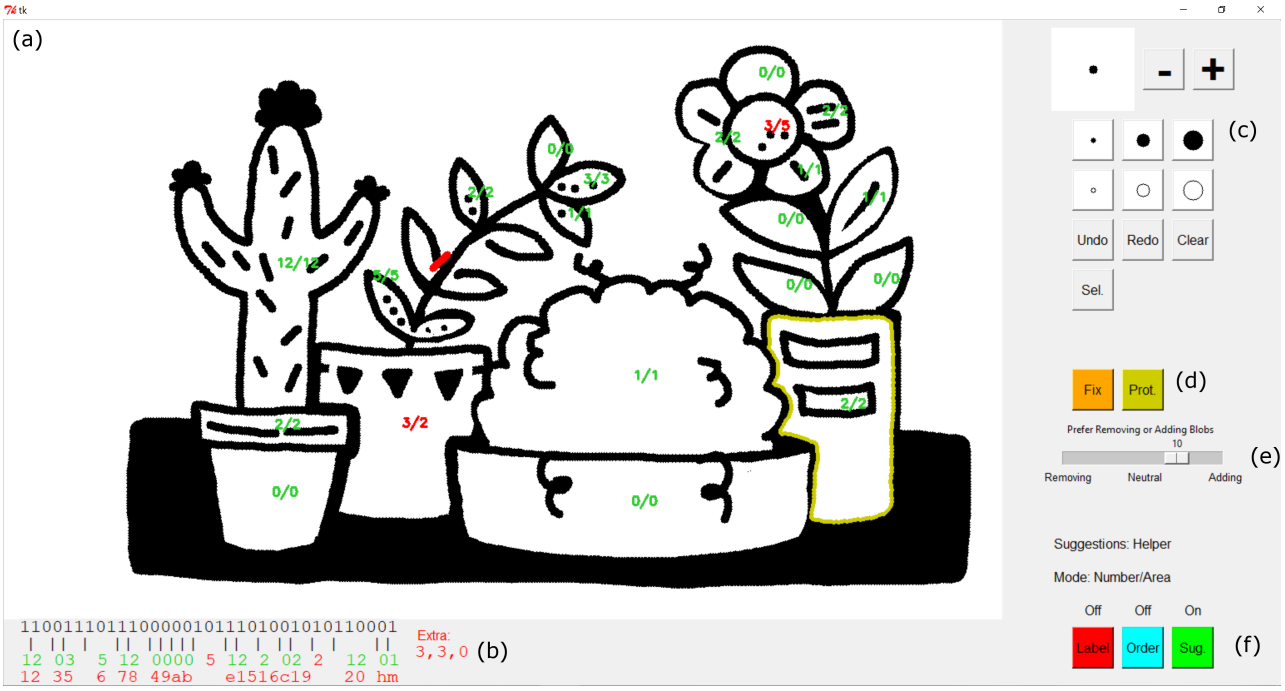

Figure 5: The drawing application used in Helper Overlay mode. (a) The canvas. (b) The target encoding panel. (c) Drawing tools. (d) Toggles for Autocomplete and Protect mode. (e) The add-remove slider. (f) Visualization feature toggle buttons.

As the marker is drawn, fractions are overlaid on top of every part indicating the current number of blobs (numerator) and the desired number of blobs (denominator), as determined by a tree-search run for 0.1 seconds. This short time limit allows the overlay to shift in real-time, updating fractions or shifting their values as new parts are drawn. These elements appear in Figure 5a, where green values indicate the current value of a part matches its desired value, and red values indicate that blobs must either be added or removed. A red line striking through the boundary of a part indicates that the part should be cut near that location.

Two additional features are provided for influencing the solution generated, via modifications to the cost function used in *FindDividers*. Applying the insight that some artists prefer to add blobs while others prefer to remove them (Section 3), we implemented an add-remove slider that establishes a relative weighting for the two kinds of alteration (Figure 5e). In the center position, the slider weights them equally, but sliding it fully to one side or the other establishes a 10 times penalty for whichever action is less preferred. This may be used in combination with a 'Protect' feature that allows individual parts to be singled out. Solutions that require blob additions/removals in protected parts incur 1000 times the cost for those alterations. *FindDividers* therefore heavily favours solutions that allow protected parts to go unchanged. Protection is indicated by a yellow outline of the part (visible in Figure 5a) and is toggled by pushing a button (Figure 5d) and clicking on the part to be protected. Both features are also available when using Autocomplete.

Autocomplete Tool

The Autocomplete method attempts to focus the artist on drawing a satisfactory picture as much as possible. There are no overlaid elements, only a button (labelled 'Fix' in Figure 5d) that applies the Autocomplete Tool, adding or removing blobs to match the encoding. This encourages the artist to fully draw a satisfying picture before interacting with the encoding scheme in any way. When this point is reached (Figure 1a), an iterative process begins in which the artist activates Autocomplete, and decides if the resulting output is workable (Figure1b). If not, they can make adjustments and activate Autocomplete again. For example, if too many blobs are being added, the artist might add more parts to achieve a better result. Alternatively, they might try adjusting the add-remove slider or using the protection feature. Once a workable solution is achieved, the artist may touch up the image in any way that does not change the number of blobs in any part (Figure1c). The result of this process is a correctly encoded marker that at no point required the artist to count blobs or adjust dividers to create.

6 EXPERIMENT

We conducted a 24-participant study to assess the usability of Helper Overlay and Autocomplete relative to the 'Manual' method of encoding used by Jung and Vogel [14].

Participants and Apparatus

We recruited 24 participants from a pool of undergraduates, graduate students, and people taking part in life drawing sessions. They were aged 18 to 69 (median 23), with 9 identifying as female and 15 as male. Out of 24 participants, 15 had taken some form of art class and 9 worked in art, design, or related positions. A digital stylus was used on a Wacom Cintiq tablet to interact with the drawing application.

Procedure

Each participant took part in a single 90 minute session consisting of 2 major phases, a 10 bit encoding phase (i.e. binary encodings of length 10) and a 20 bit encoding phase. The ordering of the Manual, Helper Overlay, and Autocomplete conditions was determined using a Latin Square. The experimental protocol was as follows.

1. An explanation of the drawing application and the general structure of a marker was given. The participant was allowed a few minutes to practice basic drawing functions.
2. The first test condition was explained and the participant was allowed a few minutes to practice the specific mechanics of that condition.
3. The participant was assigned a random 10-bit binary encoding (starting with a 1) and given up to 10 minutes to draw a marker that depicted a given topic. For all 10-bit markers, the topics given were Music or Structures.
4. The participant was asked to rate the test condition on a scale of 1-7 for ease of understanding (how well they understood the mechanics of the condition) and ease of expression (how well they could draw the picture they envisioned while using the tools provided in that condition).
5. Steps 2-4 were repeated for the remaining two conditions.
6. Steps 2-4 were repeated for all conditions with 20-bit target encodings and the topics of Sports or Transportation.
7. The participant rated the overall usefulness of each method for facilitating their drawing on a scale of 1-7, and gave ordered preferences over the conditions with explanation.

Results

Ranking. By a Condorcet ranking metric [21] applied to the participants' ordered preferences, we find that Helper Overlay was the most preferred condition, Autocomplete the second, and Manual the least preferred condition. (i.e. Helper Overlay won both pairwise 'elections' and Autocomplete won its pairwise election against Manual.) Between Helper Overlay and Autocomplete, the vote was close, at 13-11 in favour of Helper Overlay, but both methods defeated Manual by large margins, at 18-6 and 16-8 for Helper Overlay and Autocomplete, respectively.

Table 1: Study summary statistics.

Cond.	Rank	Suc.% 10-bit	Suc.% 20-bit	Time (s) 10-bit	Time (s) 20-bit
Helper	1	100	92	306 [153]	394 [132]
Auto	2	96	96	327 [144]	444 [134]
Manual	3	96	71	377 [142]	512 [97]

Success Rate. We counted the number of 'successful' markers, where success is defined as a correct encoding completed within the allotted 10 minutes. Participants were not allowed to submit a marker with an incorrect encoding, so every failure occurred because time ran out. For 10-bit encodings, we observed nearly universal success across all three conditions, while for 20-bit encodings, we found success rates of 92% and 96% for Helper Overlay and Autocomplete, but only 71% for the Manual condition (summarised in Table 1). We performed Fisher's exact test for these values and determined that the difference between the Manual and Autocomplete meets the threshold for statistical significance ($p < 0.05$), while the difference between Manual and Helper Overlay falls short ($p = 0.13$). A selection of markers successfully completed by participants is given by Figure 6.

Time to Completion. Average time to completion (where failure was counted as a full 10 minutes) was calculated and found to be similar (i.e. not statistically significantly different) for all three conditions for 10-bit encodings. However, there was a larger difference for 20-bit encodings, with average times of 394, 444, and 512 seconds for Helper Overlay, Autocomplete, and Manual, respectively. (These values, along with their standard deviations, appear in Table 1.) An ANOVA with Tukey HSD post-hoc test shows the times for Helper Overlay and Manual to be significantly different ($p < 0.01$), but does not show statistical significance for the difference in times for the remaining condition pairs.

Ratings. For our analysis of the 'ease of understanding', 'ease of expression', and overall 'usefulness' questions, we examine the ratings (on a scale of 1-7) made after drawing 20-bit markers. For each rating category of each participant, we find the mean score across the 3 conditions. We then subtract this value from the individual ratings, shifting the distribution so that its mean is centered over 0. This allows for a better comparison across participants who may have had different baseline ratings. Average values are reported in Table 2. Comparing these values between conditions does not yield significant results. We also find the degree of correlation of understanding and expression with usefulness (since usefulness is more granular than ordered preferences).

Figure 6: Sample markers drawn by participants. 10-bit drawings are in the top row, and 20-bit in the bottom. From left to right, the conditions used are Manual, Helper Overlay, Autocomplete.

Table 2: Questionnaire 20-bit rating statistics.

Cond.	Understanding	Expression	Usefulness
Helper	0.12 [0.50]	0.08 [0.86]	0.41 [0.75]
Auto	0.17 [0.87]	0.29 [1.18]	0.33 [1.34]
Manual	-0.29 [0.83]	-0.38 [1.13]	-0.75 [1.55]

For 20-bit values, we find that understanding produces a Pearson correlation coefficient between 0.59 and 0.82 over a 95% confidence interval, while expression produces between 0.68 and 0.86. This indicates that both are heavily correlated with usefulness, although Pearson and Filon's z-test implies that we cannot say expression is necessarily more correlated than understanding ($p = 0.10$).

We additionally tested, but did not find significant evidence for correlations between ratings and demographic factors like art background, gender, or age.

7 DISCUSSION

We began with the hypothesis that users would be willing to give up direct control over the 'divider' portion of encoding if it allowed them to focus on the artistically creative aspects of drawing markers. For a majority of study participants, this turned out to be true. The Manual method ranked last by a wide margin. It was, however, not universally true; 5 participants selected Manual as their favourite method. For these participants, we observed a common desire for increased control expressed in the written portion of the questionnaire. In their ranking justifications, they noted that Manual encoding enabled planning, and found our automated methods to be "inflexible" or "too simplistic". On the other hand, the majority of users enjoyed that simplicity, commenting that the Manual condition was "inconvenient" or "mentally taxing".

Among participants who preferred an automated method, those who ranked Helper Overlay as first appreciated that it "allows for more freedom" and acted as a "guide" or a "recommendation engine like service". Those who chose Autocomplete noted that it was the "easiest", "created a more relaxed environment", and that it "was helpful when stuck". One participant noted that it was "very intuitive and it helped optimize creativity".

On the continuum ranging from most directly controlled by the user to most automated, Manual is on one extreme end and Autocomplete on the other. The Helper Overlay seems to provide a useful compromise, which is reinforced by the fact that only 2 participants out of 24 ranked it as their least preferred method. It is also relevant to note that the Helper Overlay and Autocomplete methods need not be mutually exclusive. For the purposes of our study, we isolated them to measure their independent effects, but in a real system, an artist might use the Helper Overlay for real-time feedback and periodically also use Autocomplete.

Autocomplete has significant room to advance. The random placement of blobs within a part is not ideal, but more aesthetically appropriate placements are heavily dependent on context. This problem is potentially a good candidate for the application of convolutional neural nets, which excel at detecting patterns (like good blob placements) in images.

Neural nets could also be used for a complete 'first pass', producing an image closer to the target encoding that could then be corrected by our current Autocomplete algorithm.

There is also room for both automated methods to better incorporate personal preference into the cost function associated with making changes to a partially complete marker. For example, we assumed that making a small cut in the boundary of a part was an action innocuous enough to warrant zero cost. However, two participants noted that, for aesthetic reasons, they specifically disliked having their parts broken. This observation suggests that our assumption was not correct for all users. In the future, preferences over the cost of part cutting may be captured by the addition of another slider, which would make it straightforward to incorporate this cost into the tree-search process.

Such a feature would necessarily be intended for expert users, as it requires well developed preferences and significant familiarity with the drawing application. This follows from low usage of the similar add-remove slider, which was used by only one participant in a 20-bit drawing. By comparison, the protect feature was used by 38% (9/24) of participants under the 20-bit Helper condition (its primary use case, due to real-time feedback), and by 17% (4/24) under the 20-bit Autocomplete condition. Both the protect and slider features are used under similar scenarios, in which the user rejects the default suggestion, but the impact of the protect feature tends to be more immediately obvious. We therefore hypothesize that more practiced users will be able to make greater use of the slider tool, as its relatively nuanced effects require experience to use proficiently. However, we also recognize that it is a good candidate for future UI improvements.

Future Work

A broader look at future work might consider branching into physical media. It has already been shown that d-touch may be effectively incorporated into large-scale artwork [19] and even ceramics [15]. In such settings, the physical medium could be continually tracked by a camera, and our tools utilized via suggestions displayed on a screen or projected directly onto the surface. While real-world readability presents additional challenges, enforcing robustness to simulated blur offers a solution. This condition, used by Costanza and Huang [7] requires that a marker maintain its encoding when shrunk to some smaller resolution. Under such conditions, it may be necessary to increase the size of shapes added and cuts made by Autocomplete. Additional study is required to determine how this might impact user interaction.

Effects on aesthetic quality, as judged by others, could also be examined in either physical or digital media. Such a study would require two groups of participants: one to draw markers under various conditions, and one to rank the resulting markers according to their subjective aesthetic preferences. If a tool was found to produce a neutral average effect on ranking, it would indicate that an artist could choose to make use of it without fear of aesthetic drawbacks. A positive average effect would suggest that its use should be actively encouraged.

8 CONCLUSION

We have developed two methods for the intentional encoding of human-designable markers that simplify the encoding process, and allow the artist to focus on producing aesthetically pleasing work. In a lab study, they were both found to be preferable to the previous method of Manual encoding by a majority of participants. The most popular method, Helper Overlay, guides the user, striking a balance between simplicity and artistic freedom. Autocomplete takes more control, but eliminates interaction with the encoding almost entirely, allowing the user to focus on drawing. Human-designable markers are a technology that is intended to be used by artists rather than programmers. For them to adopt this approach, it is essential to develop tools that privilege creativity over technical details.

9 ACKNOWLEDGEMENTS

We would like to thank Erin Jung for drawing the markers pictured in Figures 1 and 5. This work was made possible by NSERC Discovery Grant #2018-05187, the Canada Foundation for Innovation Infrastructure Fund "Facility for Fully Interactive Physio-digital Spaces" (#33151), and Ontario Early Researcher Award #ER16-12-184.

REFERENCES

[1] 2017. TinyURL. (2017). http://tinyurl.com/

[2] Alper T Alan, Mike Shann, Enrico Costanza, Sarvapali D Ramchurn, and Sven Seuken. 2016. It is too hot: An in-situ study of three designs for heating. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 5262–5273.

[3] R. Bencina, S. Jorda, and M. Kaltenbrunner. 2005. Improved Topological Fiducial Tracking in the reacTIVision System. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops(CVPRW)*, Vol. 00. 99. https://doi.org/10.1109/CVPR.2005.475

[4] Margaret A Boden and Ernest A Edmonds. 2009. What is generative art? *Digital Creativity* 20, 1-2 (2009), 21–46.

[5] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[7] Enrico Costanza and Jeffrey Huang. 2009. Designable visual markers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1879–1888.

[8] Enrico Costanza and Mirja Leinss. 2006. Telling a story on a tag: the importance of markers' visual design for real world applications. In *Workshop Mobile Interaction with the Real World (MIRW 2006) in conjunction with 8th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI 2006), Espoo, Finland*. Citeseer.

[9] Enrico Costanza and John Robinson. 2003. A Region Adjacency Tree Approach to the Detection and Design of Fiducials. (2003).

[10] David Ha and Douglas Eck. 2017. A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477* (2017).

[11] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 159–166.

[12] ISO. 2015. *Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification.* ISO 18004:2015. International Organization for Standardization, Geneva, Switzerland.

[13] David J Johnston and Adrian F Clark. 2003. A Vision-Based Location System using Fiducials. In *VVG*. 159–166.

[14] Joshua DA Jung and Daniel Vogel. 2018. Methods for Intentional Encoding of High Capacity Human-Designable Visual Markers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 313.

[15] Rupert Meese, Shakir Ali, Emily-Clare Thorne, Steve D Benford, Anthony Quinn, Richard Mortier, Boriana N Koleva, Tony Pridmore, and Sharon L Baurley. 2013. From codes to patterns: designing interactive decoration for tableware. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 931–940.

[16] Gonzalo Gabriel Méndez, Miguel A Nacenta, and Uta Hinrichs. 2018. Considering Agency and Data Granularity in the Design of Visualization Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 638.

[17] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM computing surveys (CSUR)* 33, 1 (2001), 31–88.

[18] William Preston, Steve Benford, Emily-Clare Thorn, Boriana Koleva, Stefan Rennick-Egglestone, Richard Mortier, Anthony Quinn, John Stell, and Michael Worboys. 2017. Enabling hand-crafted visual markers at scale. In *Proceedings of the 2017 Conference on Designing Interactive Systems*. ACM, 1227–1237.

[19] Emily-Clare Thorn, Stefan Rennick-Egglestone, Boriana Koleva, William Preston, Steve Benford, Anthony Quinn, and Richard Mortier. 2016. Exploring large-scale interactive public illustrations. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*. ACM, 17–27.

[20] Norman J Woodland and Silver Bernard. 1952. Classifying apparatus and method. (Oct. 7 1952). US Patent 2,612,994.

[21] H Peyton Young. 1988. Condorcet's theory of voting. *American Political science review* 82, 4 (1988), 1231–1244.