Textility of code: a catalogue of errors

David Griffiths and Alex McLean

February 8, 2017

1 Introduction

Through the following article, we look for different ways to represent the structure of ancient weaves with contemporary source code. Unusually for an article in the present journal, we approach this topic from a position of naivete, as computer programmers rather than textile artists. Accordingly, our aims are humble; to attempt to simulate certain aspects of weaving in order to greater understand its structure, and its relation to contemporary computer languages.

The title of this article is intended to recognise the difficulty of simulating woven structure; any attempt to fully simulate cloth is in some sense doomed to failure. The physical properties of thread at multiple scales, the duality of warp and weft, the three dimensional structures of ups and downs first under tension and then released, and the continuous and discrete interactions between thread that allow a cloth to hold together (or otherwise fall apart), and much more besides.

Textile craftspeople have developed the technology of weaving over millennia, often passing on knowledge through non-verbal means as tacit knowledge. Approaching this technology as computer scientists, we have found weaving to support a great depth of physical engagement with the fundamental nature of computation. Indeed, it would seem that the far younger field of computer science has a great deal to learn from the field of textile design. We therefore approach weaving not in order to create tools to support or even less 'solve' it, but rather to help communicate the complexity of its structure and perhaps look for ways in which textiles could inform the development of computer science.

2 Understanding plain weave

Most of the work presented here was conducted as part of the *Weaving Codes, Coding Weaves* project, funded by the Arts and Humanities Research Council between 2014 and 2016. However, our initial attempt at reaching an understanding of the complexities of weaving took place during the Mathematickal arts workshop at FoAM Brussels in 2011. This workshop, with Tim Boykett and textile designer and educator Carole Collet was devoted to bringing together the arts of mathematics, textiles and computer programming.

Plain (or tabby) weave is the simplest woven structure, but nonetheless, when woven with sequences of coloured warp and weft threads can produce a great variety of pattern. It only takes a few lines of code (shown below in the *Scheme* language) to calculate the colours of a plain weave, using lists of warp and weft yarn as input.

```
;; return warp or weft, dependent on the position
(define (stitch x y warp weft)
  ;; a simple way to describe a 2x2 plain weave kernel
  (if (eq? (modulo x 2)
           (modulo y 2))
  warp weft))

;; prints out a weaving, picking warp or weft depending on the
;; position
(define (weave warp weft)
  (for ((x (in-range 0 (length weft))))
     (for ((y (in-range 0 (length warp))))
        (display (stitch x y
                         (list-ref warp y)
                         (list-ref weft x))))
   (newline)))
```

With this small computer program we may visualise the weaves with textual symbols representing different colours. For example, to specify distinct colours for the warp and weft threads:

```
(weave '(0 0 0 0 0 0 0) '(: : : : : : : :))

     0 : 0 : 0 : 0
     : 0 : 0 : 0 :
     0 : 0 : 0 : 0
     : 0 : 0 : 0 :
     0 : 0 : 0 : 0
     : 0 : 0 : 0 :
     0 : 0 : 0 : 0
     : 0 : 0 : 0 :
     0 : 0 : 0 : 0
```

The above simply shows the structure of the warp/weft crossings, with all warps having a colour represented by `0`, and all wefts having one represented by `:`. With slightly more complex colourings, we quickly got a glimpse of the generative possibilities of even plain weave. For example, 2:2 alternating colour of both warp and weft threads, with an offset on the weft:

```
(weave '(0 0 : : 0 0 : : 0 0) '(0 : : 0 0 : : 0 0 :))

     : 0 : : : 0 : : : 0
     0 : : : 0 : : : 0 :
     0 0 0 : 0 0 0 : 0 0
     0 0 : 0 0 0 : 0 0 0
     : 0 : : : 0 : : : 0
     0 : : : 0 : : : 0 :
     0 0 0 : 0 0 0 : 0 0
     0 0 : 0 0 0 : 0 0 0
     : 0 : : : 0 : : : 0
```

This emergence of pattern will be familiar to an experienced weaver, but potentially of great surprise to a computer programmer. We wanted to explore these warp/weft thread colour patterns of plain weave further, by generating them algorithmically. We chose Lindenmayer systems (L-systems), which are formal grammars originally used to model plant

or cellular growth (See e.g. Flake 2000). L-systems can be related to weaves, in that they consist of rules which may appear to be simple, but which often generate complex results which come as a surprise to the uninitiated. We began with a starting colour (known as an *axiom*), and then followed two 'search-replace' operations repeatedly, following the following simple rules:

```
Axiom: O
Rule 1: O ⇒ O:O:
Rule 2: : ⇒ :O:
```

In the above, ⇒ simply means search for the symbol on the left, and replace with the symbols on the right. So, we begin with the axiom:

```
O
```

Then run rule 1 on it - replacing `O` with `O:O:`

```
O:O:
```

Then run rule two, replacing all instances of `:` with `:O::`

```
O:O:O:O:
```

And repeat both these steps one more time:

```
O:O:O:O::O:O:O:O:O::O:O:O:O:O::O:O:O:O:O::O:
```

This technique allows us to use a very small representation that expands into a long, complex form. However, our simple text-based representation does not fully represent how the weave appears and behaves as a three-dimensional textile, so our next step was to try weaving these patterns.

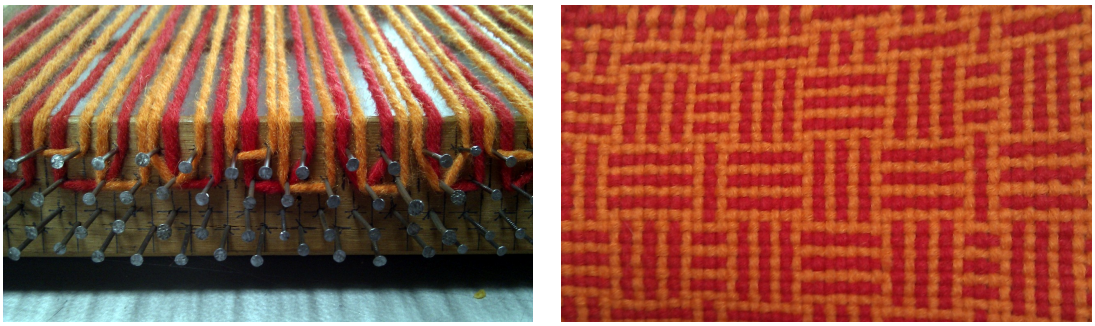
We could keep running our rules forever, the string of text will just keep growing. However, to create a fabric of manageable size we decided to run them just one more time, then read off the pattern replacing `O` for red and `:` as orange to warp a frame loom, shown in Figure 1a. When weaving, we followed the same sequence for the weft threads, resulting in the textile shown in Figure 1b.

There were two motivations behind this approach, firstly to begin to understand weaving by modelling plain weave, and confirming a hypothesis (the text-based version of the pattern) by following instructions produced by the language to actually weave them. The other aspect was to use a generative formal grammar to explore the patterns possible given the restriction of plain weave, perhaps in a different manner to that used by weavers – but one that starts to treat weaving as a computational medium.

This system was restricted by only working with plain weave. Given the range of patterns possible, this was not of immediate concern, but something we addressed in our later systems.

3 Four shaft loom simulation

In order to engage further with the complexities of woven structure, we needed to look beyond plain weave. Our next step was to model the workings of a four-shaft loom. One of our core



(a) The warped frame loom.

(b) Close-up of resulting weave.

Figure 1: Realisation of weave resulting from a pattern generated from an L-System.

aims is to gain knowledge about the computational processes involved in weaving, and so we need to gain deep understanding of how a loom works. As a result, our four-shaft loom simulation takes a different approach to that of most weaving simulation software. In much contemporary weaving software, you 'draw' a two-dimensional image, and the software then generates instructions for how it may be woven. However, our simulation does the opposite - you describe the set up of the loom, and the software tells you what visual patterns result. Our latter approach brings the three-dimensional structure of a weave to the fore, allowing us to investigate the complex interference patterns of warp and weft, considering the visual result in close relation to the underlying structure that it arises from.

Our model of a four-shaft loom calculates the *shed* (the gap between ordered warp threads), processing each shaft in turn and using a logical "OR" operation on each warp thread to calculate which ones are picked up. This turns out to be the core of the algorithm – a key excerpt of which is shown below:

```
;; 'or's two lists together:
;; (list-or (list 0 1 1 0) (list 0 0 1 1)) ⇒ (list 0 1 1 1)
(define (list-or a b)
  (map2
   (lambda (a b)
     (if (or (not (zero? a)) (not (zero? b))) 1 0))
   a b))

;; calculate the shed, given a lift plan position counter
;; shed is 0/1 for each warp thread: up/down
(define (loom-shed l lift-counter)
  (foldl
   (lambda (a b)
     (list-or a b))
   (build-list (length (car (loom-heddles l))) (lambda (a) 0))
   (loom-heddles-raised l lift-counter)))
```

This code embodies understanding of how the warping of a loom corresponds to the patterns it produces, and may be quickly experimented and played with in a way that is not possible on a physical loom (which would require time consuming re-warping with each change). Figure 2 shows an example weave, with both warp and the weft alternating between light and dark yarns.

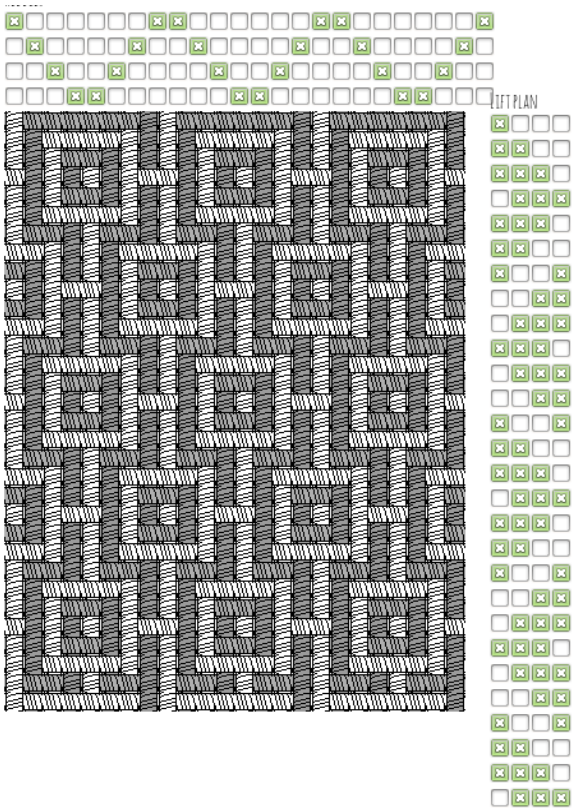
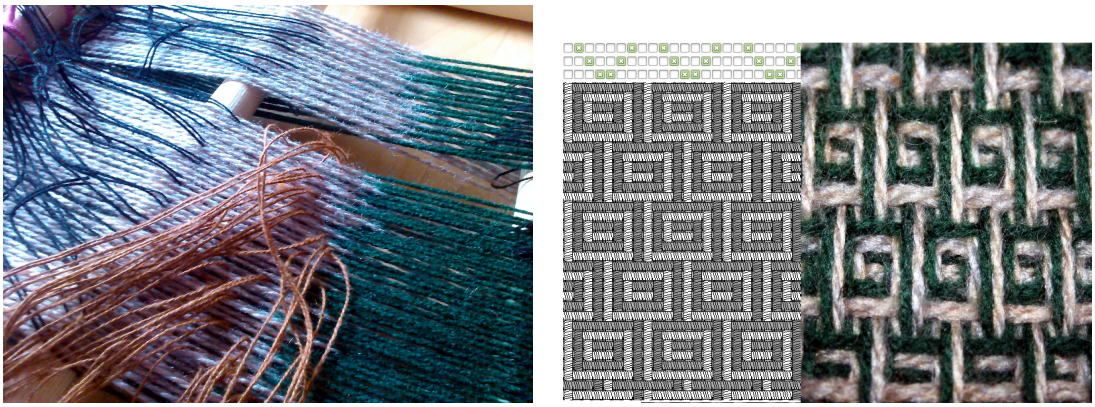


Figure 2: The interface for our four shaft loom simulation, showing heddles (above), lift plan (to the right) and simulated weave.

As with testing our understanding of plain weave in the previous section, our next step was to try weaving the structures with a real loom and threads, in order to test our model. A frame loom was constructed to weave these patterns, pictured in Figure 3a. Here the shafts are sleyed to pick up the warp as defined by the simulation's input (the checkboxes seen in Fig. 2). The threads (which form heddles) are tied on to wooden poles which are pulled in different combinations during weaving. This is a similar technique to that used in warp weighted looms, and is much faster than counting threads manually each time. It is important to construct heddles from thinner threads than the warp, but as they are put under tension during the weaving process they do need to be strong. Fittingly for our Weaving Codes project, configuring the warp with heddles felt very much like coding threads, with threads.

Often, an improvised approach to weaving is called for, even when using a predefined pattern. There is a great deal of reasoning required to respond to issues of structure that cannot be fully anticipated. This is because you need to respond to the interactions between the materials and the loom itself. The clearest example is the *selvedge* or edges of the fabric, which must be thought about and solved 'live' as you weave. In order to keep the weave from falling apart you need to 'tweak' the first and last warp thread based on which weft yarn colour thread you are using. The different weft threads also need to go over/under each other in a suitable manner.

In relation to computer programming, this improvisation at the loom is analogous to *live*



(a) Frame loom constructed to test our simulation

(b) Close-ups of a simulated *meander* pattern, and its actual weave (with floating threads)

Figure 3: Testing our simulation of a four-shaft loom

coding, where code is written 'on the fly', often as a performing art such as live music making (Blackwell et al. 2014). See Emma Cocker's article in the present issue of Textile for deep investigation into the relation between between live weaving and live coding. ** Please insert reference **

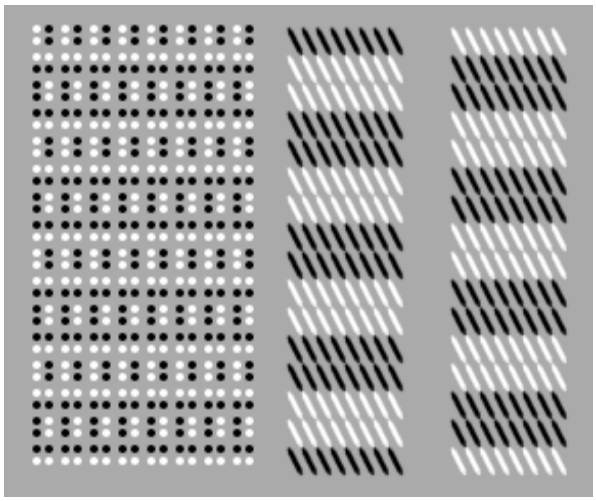
Figure 3b shows close-ups of the simulated *meander* pattern and its actual weave. There are clear differences visible between them, due to the behaviour of the long 'floating' threads; the pattern would be distorted further if the fabric were removed from the loom and the tension lost. The extent to which it is possible, or even desirable to include such material limitations into a weaving language or model was one of our main topics of inquiry when talking to our advisers (particularly esteemed industrial weaver Leslie Downes). Through discussion, we came to understand that simulating such physical interaction between threads is beyond even the even the most expensive simulation software.

We have already mentioned two aspects of weaving which do not feature in weaving software; the structure of selvedge, and the behaviour of floats. We also noted a third, more subtle limitation: some sequences of sheds cause problems when packing down the weft, for example if you are not too careful you can cause the neat ordering of the weft colours to be disrupted in some situations, where in practice they overlap.

4 Tablet weaving simulation

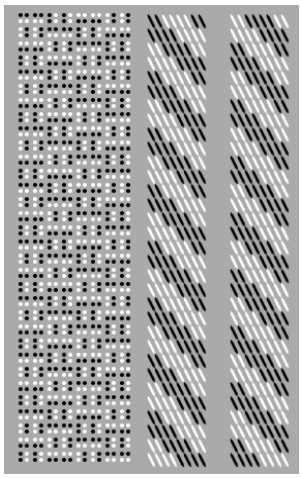
Tablet weaving (also known as card weaving) is an ancient form of textile pattern production, using cards which are rotated to provide different sheds between warp threads. This technique produces long strips of fabric, which in antiquity were used as starting bands, forming the borders of a larger warp weighted weaving.

Tablet weaving is extremely complex, so in a similar manner to the 4 shaft loom, we devised a language/notation to help better understand it. As before, this language can be used either to drive a computer simulation, or can be followed when weaving. The following shows the output of a simple program written in this language, consisting of the single-instruction procedure `(weave-forward 16)`.



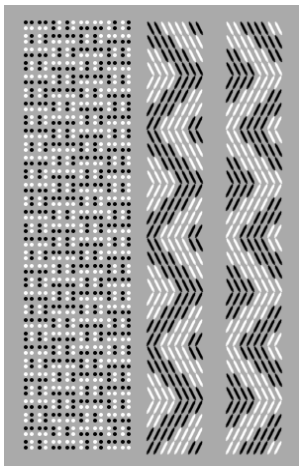
The language consists of such simple instructions to represent the movement of the cards to create each shed. The previous example shows a simple case where cards are moved a quarter-turn to create each of 16 sheds. The card rotations are shown on the left for each of 8 cards, the simulated weaving is on the right for the top and bottom of the fabric. The `(weave-forward 16)` instruction repeats the operation of moving all the cards a quarter turn and adding a weft, a total of 16 times.

In our simulation, the cards are set up with a double face weave on square cards, with threads coloured (clockwise from the top right corner) black, black, white, white. We can offset the rotation of these cards from each other first, to change the pattern. The `rotate-forward` instruction turns only the specified cards a quarter turn forward without weaving a weft, illustrated in the following code and simulated output:



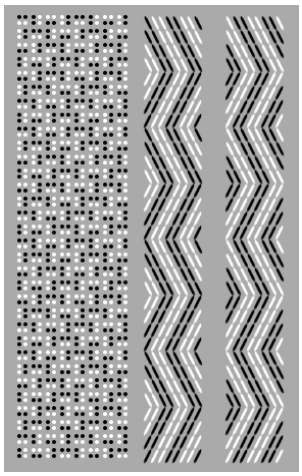
```
(rotate-forward 0 1 2 3 4 5)
(rotate-forward 0 1 2 3)
(rotate-forward 0 1)
(weave-forward 32)
```

One aspect of tablet weaving not included in our simulation is that it is not physically possible to weave such high numbers of forward rotations without completely twisting up the warp. In practice, we need to balance forwards and backwards movements to make something physically weavable. However if we do so, then a 'zig zag' pattern results, as the below demonstrates.



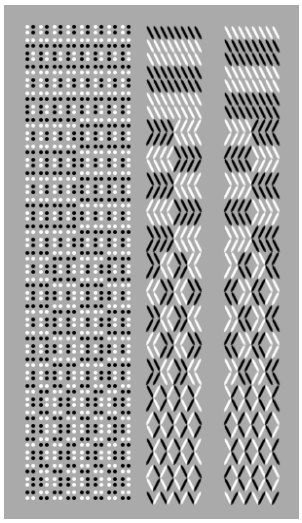
```
(rotate-forward 0 1 2 3 4 5)
(rotate-forward 0 1 2 3)
(rotate-forward 0 1)
(repeat 4
  (weave-forward 4)
  (weave-back 4))
```

The following shows that a different starting pattern better matches the stitch direction.



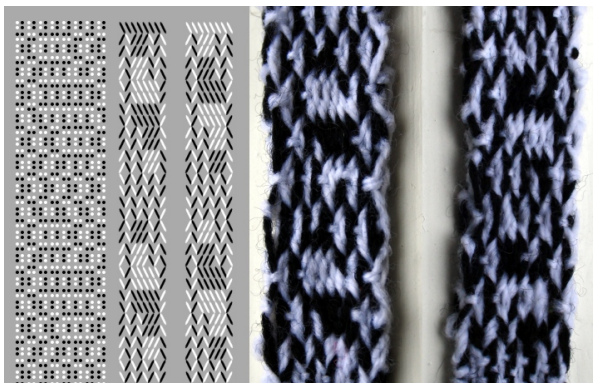
```
(rotate-forward 0 1 2 3 4 5 6)
(rotate-forward 0 1 2 3 4 5)
(rotate-forward 0 1 2 3 4)
(rotate-forward 0 1 2 3)
(rotate-forward 0 1 2)
(rotate-forward 0 1)
(rotate-forward 0)
(repeat 4
  (weave-forward 4)
  (weave-back 4))
```

Another way of forming patterns is by *twisting* the cards. The `twist` command takes a list of cards to twist, and results in these cards reversing their turn direction compared to the others. With double faced weave, the twist needs to take place when the cards are in the right rotation, otherwise we get an 'error', such as that shown below:



```
(weave-forward 7)
    (twist 0 1 2 3)
    (weave-back 1)
    (repeat 2
      (weave-forward 2)
      (weave-back 2))
    (weave-forward 1)
    (twist 2 3 4 5)
    (weave-back 1)
    (repeat 2
      (weave-forward 2)
      (weave-back 2))
    (weave-forward 1)
    (twist 1 2 5 6)
    (weave-back 1)
    (repeat 2
      (weave-forward 2)
      (weave-back 2))
```

If we put our encoded twists into repeating loops, we can make small programs with complex results. You can see a comparison between the simulated and woven form below, the latter created by following the program by hand.



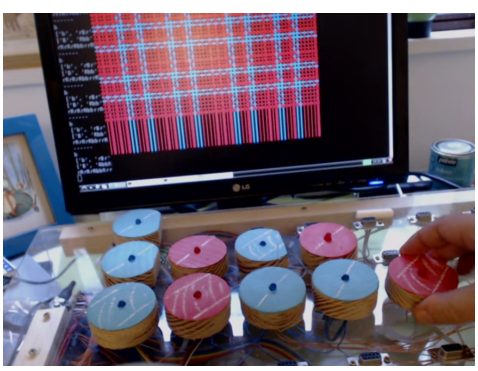
```
(weave-forward 1)
    (twist 0 2 4 6)
    (repeat 4
      (twist 3)
      (weave-forward 4)
      (twist 5)
      (weave-back 4))
```

This tablet weaving language was the first we created that describes the actions and movements of the weaver. It was mainly of use in understanding the complexities of tablet weaving, however some of this still remains a mystery – the calculation of the inverse side of the weaving is not correct, probably due to the double twining of the weave. In some cases it has very different results, in others it matches perfectly. Further experimentation is needed.

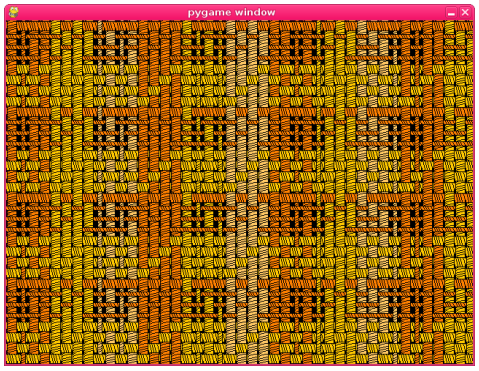
This language also began our investigations into combining tablet and warp weighted weaving techniques into a single notation system. This remains an ongoing challenge, but pointed in the direction of a more general approach being required - rather than either a loom or weaver-centric view.

5 Flotsam Raspberry Pi Simulation

Since antiquity, weaving has been a fundamentally *digital* technology, as it involves combinations of discrete elements or threads. For this reason we place it in the same category as modern digital technology which involves combinations of discrete voltages – computers,



(a) Interface designed for teaching children programming.



(b) A woven pattern created with Flotsam, exhibiting different threads qualities described using the physical blocks.

Figure 4: The Flotsam prototype

smartphones and so on. When we do this we see many differences between the design of these tools, how we interact with them, and their relationship to our bodies. One of the important strands of research on our project turned out to be looking for how the design of weaving tools, having been honed over very many generations and across many cultures – can inform the design of programming tools and help us with some of their limitations.

Flotsam is a prototype, screen-less tangible programming language largely built from driftwood. We constructed it in order to experiment with new types of "tangible hardware", for teaching children programming, without the need for a traditional keyboard-and-screen, single user interface. It is based on the same L-system as was used for the first mathematickal arts workshop (see §2), and describes weave structure and patterns using wooden blocks representing yarn width and colour. The L-system rules for the warp/weft yarn sequences are constructed from the positions the blocks are plugged into, using a custom hardware interface.

The weaving simulation runs on a Raspberry Pi computer, and the overall system is designed to describe weave patterns which include yarn properties beyond colour, which are therefore beyond the capability of Jacquard looms. The version shown in Figure 4a is restricted to plain weave, but more complex structures can be created as Figure 4b demonstrates. The flotsam tangible hardware was used in primary schools and tutoring with children, and was designed so the blocks could be used in many different ways – for example, experiments beyond weaving included an interface with the popular *Minecraft* computer game and a music synthesiser.

As before, the L-system programming approach provided quick exploration of the huge variety of weaving patterns, although this approach perhaps distracts from one of our core goals - to understand and communicate how weavers think. The physical design of the system itself needed further development, as the plugs were tricky to position correctly - particularly for small fingers. However, during use we related strongly to the findings of Horn et al. (2009), in that the tangible interface appeared to encourage collaborative learning beyond that possible with a traditional keyboard and screen interface designed for a single user.

Another possibility with this kind of physical interface is the increased role of *touch* – we wrapped the tokens with the kinds of yarn that they represented (and the 'replication' tokens

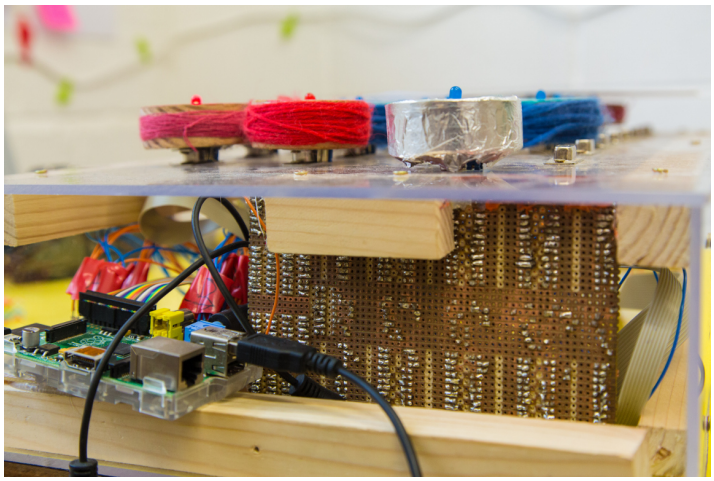


Figure 5: Profile of the Flotsam prototype, with tactile blocks in situ.

in tinfoil), as a way to allow people to *feel* the symbolic representation rather than needing to see it, increasing the range of senses in use during programming as well as making the system much easier to explain.

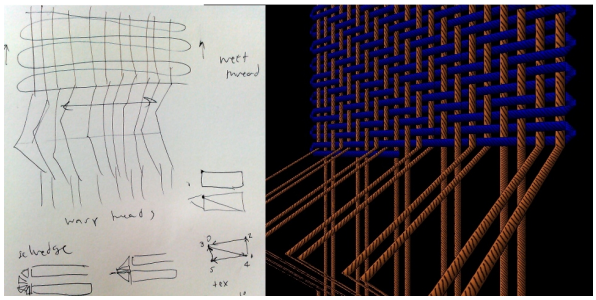
6 Pattern matrix warp weighted loom simulation

One of the main objectives of our project was to provide a simulation of the warp weighted loom to use in demonstrations, in order to explore and share ancient weaving techniques. Beyond our previous simulations we needed to show the actual weaving process, rather than the end result, in order to explain how the structures and patterns emerge. Weaving is very much a 3-dimensional process, and our previous visualisations failed to show this well.

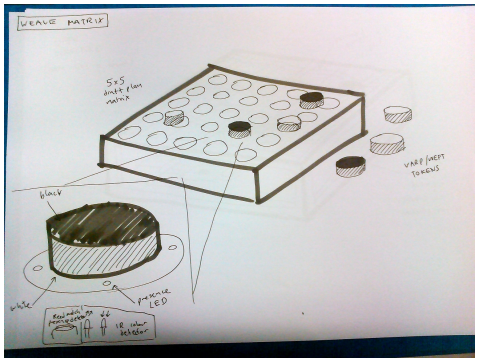
We built a 3D simulation of a warp weighted loom which ran on a Raspberry Pi computer, which allows for easy integration with our experimental hardware.

Following our experience with the Flotsam prototype, we decided to explore tangible programming further. The pattern matrix was the next step, specialised for weaving and built by Makernow (Oliver Hatfield, Andrew Smith, Justin Marshall; `http://www.makernow.co.uk/`) and FoAM Kernow. The pattern matrix was initially designed for use in Miners Court, an extra care housing scheme in Redruth, Cornwall alongside other crafts and technology workshops as part of the Future Thinking for Social Living (`http://ft4sl.tumblr.com/`) project with Falmouth University. This interface was developed further in a public setting, during a project residency at Museum fr Abgsse Klassischer Bildwerke (Museum of Casts of Classical Sculpture) in Munich.

A technical challenge for the pattern matrix was to remove the need for physical plugs, which proved problematic with the Flotsam prototype. The affordability of the programming blocks themselves was also an important constraint, partly due to the need for use in public places. We therefore designed the blocks as disks, painted white and black on opposing sides, and without physical connections. Instead, we embedded used magnets within them so that the orientation and position of the disks could be picked up by a grid of sensors in the base of the pattern matrix, via the *hall effect*. The hall effect sensors detect the polarity of nearby



(a) Sketch by Ellen Harlizius-Klck while explaining the warp weighted loom (left) and resulting simulation (right).



(b) The *pattern matrix* design.

Figure 6: Design sketches that lead to the *Pattern Matrix*

magnetic fields, and even with fairly weak magnets we found we could put the sensors right next to each other and still determine the difference between two opposed or aligned fields.

For the warp/weft weave pattern structure, we only need a single 'bit' value to be detected per block, where on / off corresponds to over or under. However for other features, such as yarn colour selection, we needed to be able to represent more information, so allowed for four bits to be encoded through the magnet alignments. Accordingly, we used four hall effect sensors in a square, allowing us to detect rotation and flipping of the blocks. At this point, we noticed that this has parallels with tablet weaving – both in terms of the notation, and the flipping and rotation actions required to use the device. We found that we can represent all sixteen possible states with only four blocks – if negative is `0` and positive is `1`, and we read the code as binary numbers, clockwise from top left. The following shows the four different magnet configurations we used in the blocks, how they change their states with twisting and flipping, and the decimal numbers these states represent:-

Starting state - decimal values: 0,1,5,6

```
- -   + -   + -   - +
- -   - -   - +   - +
```

Rotate clockwise - decimal values: 0,2,10,12

```
- -   - +   - +   - -
- -   - -   + -   + +
```

Horizontal flip - decimal values: 15,11,10,12

```
+ +   + +   - +   - -
+ +   + -   + -   + +
```

Rotate counter-clockwise - decimal values: 15,13,5,6

```
+ +   + -   + -   - +
+ +   + +   - +   - +
```

Figure 7: A member of staff at Miners Court extra care housing scheme trying the first working version of the tangible weavecoding. The Raspberry Pi displays the weave structure on the simulated warp weighted loom, with a single colour for each warp and weft thread.

Vertical flip - decimal values: 0,4,5,6

```
- -   - -   + -   - +
- -   - +   - +   - +
```

The 3D warp weighted loom simulation was our first to include selvedge calculation, as well as animating the shed lift and weft thread movement. The inclusion of the selvedge, along with multiple weft threads for the colour patterns, meant that the possibilities for the selvedge structure was very high. We don't yet have a way to notate these possibilities, but at least we could finally visualise this, and the simulation could be used to explain complexities in this ancient weaving technique.

7 Representing thread

A central puzzle to our project is how to represent woven thread, using code. In the following we will share examples written using the Haskell programming language, which supports careful thinking of structure, in terms of the *types* of data in a language. Many readers will not be familiar with this programming language, but we include them to at least share a trace of our thinking around the structure of woven threads.

A naive approach to representing a weave would be as a two-dimensional grid or raster, for example as a list of lists of boolean values, representing ups and downs:

```
data Weave = Weave {draft :: [[Bool]]}
```

This allows the woven structure within a weaver's draft to be represented, and fits programming language affordances very well, which are well suited to the processing of lists. But this provides a very particular point of view, which does not consider the path of a thread, including at the selvedge. Our question then was, what happens if we attempt to find a represent which takes the point of view of a thread, rather than a weaver?

The below lists just some of the different ways we thought about representing a weave. Each represents a different point of view or way of thinking, in terms of movement of thread,

loom and weaver. They do not represent a program to execute, but merely different opposing ways in which a computer could represent a thread and its weave.

```
-- Thread properties as a list
type Thread a = [a]

-- Thread properties as behaviour over its length
type Thread a = Int → a

-- Construction of a weave as change of state
type Pick = Weave → Weave

-- Construction of a weave as a sequence of movements (like a turtle)
data Move = Over | Under | TurnLeft | TurnRight
type Weave = [Move]

-- Weave as path - coordinate based
data Weave = Weave {thread :: [(Int,Int,Bool)]}

-- Structure of weft relative to state
data Change = Up | Down | Toggle | Same Int | Different Int
data Weave = Weave [Change]
tabby = Weave (Up:repeat Toggle)

-- Lift plan
data Pull = Up | Down
data HeddleRod = HeddleRod [Pull]
data Weave = Weave [[HeddleRod]]

-- Unweave
data Untangle = Pull | Unloop
data Unweave = [Untangle]

-- Discrete thread
data Direction = Left | Right
data Thread = Straight | Turn Direction | Loop Direction | Over | Under

-- Weave as action
data Direction = Left90 | Right90 | Left180 | Right180
data Action = Pull Int | Turn Direction | Over | Under
```

Through this process, the representation that we found ourselves most drawn to was that which focussed on the actions of a thread:

```
data Action = Over | Under | Pull Int | TurnIn | TurnOut
type Weave = [Action]
```

The first two actions represent a thread going *over* and *under* another thread (which may be itself, in the case where it has turned back upon itself). The third represents a thread being *pulled* a discrete number of measures, or in other words, a thread being under tension, over a distance. We felt this important, as tension is an key part of the structure of a weave, at least while it is on the loom.

The final actions are of the thread turning an assumed 90 degrees, either *in* or *out*. In practice, *in* means turn in the same direction as the previous turn, and *out* in the opposite direction. A more obvious approach would be to explicitly represent left and right turns, but this does not make sense from the perspective of a thread. The structure of a thread

is of twisted fibres, with little purchase on a left/right orientation, which only makes sense from the perspective of the weave, not the thread. This thread-centric concept of *in* and *out* operates relative to the previous turn, rather than the weave as a whole.

Happily, the concept of *in* and *out* fits well with the back-and-forth movements of woven thread, particularly a weft thread. Rather than thinking in terms of turning left on one side, and right on the other, we can think of both turns in terms of the thread first turning *out* from the weave, and then back *in* to turn back on itself on the next row. Accordingly the creation of a warp of `n` threads of `l` length can be represented as a repeating cycle of just three steps:

```
warp n l = take (n*3) $ cycle [Pull l,TurnOut,TurnIn]
```

Similarly, we can represent a plain or tabby weave with a single thread, by composing together a warp, with a turn inward, and a repeating over/under:

```
tabby h w = warp h w ++ [TurnIn]
                     ++ threadWeftBy (rot 1) ([Over,Under]) h w
```

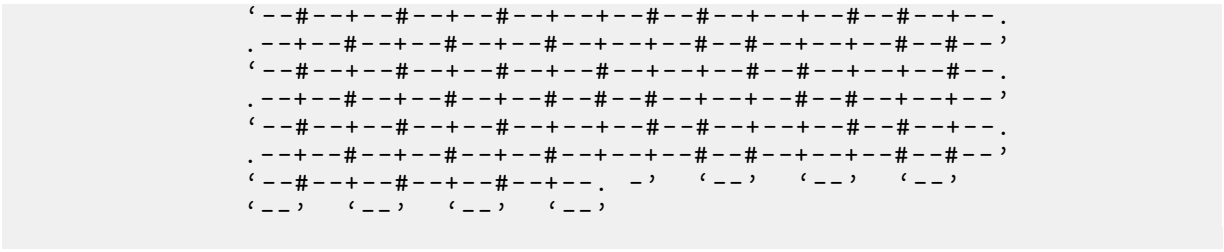
Following this weave produces the following structure, again rendered with text, where `#` represents over and `+` under.

```
 .-     .--.  .--.  .--.  .--.
 '--#--+--#--+--#--+--#--+--#--.
 .--+--#--+--#--+--#--+--#--+--'
 '--#--+--#--+--#--+--#--+--#--.
 .--+--#--+--#--+--#--+--#--+--'
 '--#--+--#--+--#--+--#--+--#--.
 .--+--#--+--#--+--#--+--#--+--'
 '--#--+--#--+--#--+--#--+--#--.
 .--+--#--+--#--+--#--+--#--+--'
 '--#--+--#--+--#--+--#--+--#--.
    '--'  '--'  '--'  '--'  '--'
```

An advantage of this approach is that it is able to represent a case where lengths of weft threads are pulled and put under tension, in order to become a warp for a later stage of the weave. This is of particular interest to our project, in relation to the ancient method of starting with a tablet woven band, with an extended weft which later becomes the warp. The following composition demonstrates such a proof of concept, where a the weft of a four-twill later becomes the warp of a tabby pattern.

```
[TurnOut,TurnIn,TurnOut,Pull 8, TurnIn,Pull 1] ++ warp 8 9 ++ [TurnIn]
 ++ (weftToWarp 6 $
     threadWeftBy'' Odd (rot 1) [Over,Over,Under,Under] 8 9
    )
 ++ [TurnOut,TurnIn,TurnIn]
 ++ threadWeftBy'' Odd (rot 1) [Over,Under] 10 6
```

```
       .--------------------------.
    '--'         .--.  .--.  .--.  |  .--.  .--.  .--.  .--.
              .--+--#--+--#--+--#--+--+--#--#--+--+--#--#--'
              '--#--+--#--+--#--+--#--+--#--#--+--+--#--+--.
              .--+--#--+--#--+--#--#--+--+--#--#--+--+--+--'
```



8 Conclusion

Conclusion goes here.

Bibliography

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber. 2014. "Collaboration and learning through live coding (Dagstuhl Seminar 13382)." Edited by Alan Blackwell, Alex McLean, James Noble, and Julian Rohrhuber. *Dagstuhl Reports* 3 (9): 130–168.

Flake, Gary W. 2000. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. Reprint. Paperback; A Bradford Book.

Horn, Michael S., Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. Jacob. 2009. "Comparing the Use of Tangible and Graphical Programming Languages for Informal Science Education." In *Proceedings of the Sigchi Conference on Human Factors in Computing Systems*, 975–984. CHI '09. New York, NY, USA: ACM. doi:10.1145/1518701.1518851. `http://doi.acm.org/10.1145/1518701.1518851`.