Textility of code: a catalogue of errors

David Griffiths and Alex McLean

January 25, 2017

1 Introduction

Through the following article, we look for different ways to represent the structure of ancient weaves with contemporary source code, and reflect upon the long history of such efforts, going back to the Euclid's Book of Elements.

2 Understanding plain weave

Our initial attempt at reaching an understanding of the complexities of weaving predated the WeavingCodes project and 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 when combined with sequences of colour it can produce many different types of pattern. For example, some of these patterns when combined with muted colours, have in the past been used as a type of camouflage – and are classified into District Checks for use in hunting in Lowland Scotland.

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 '(O O O O O O O) '(: : : : : : : :))

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

The above simply shows the structure of the warp/weft crossings, with all warps having a colour represented by `O`, and all wefts by 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 '(O O : : O O : : O O) '(O : : O O : : O O :))

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

This emergence of pattern will be familiar to an experienced weaver, but a 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. 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:`:

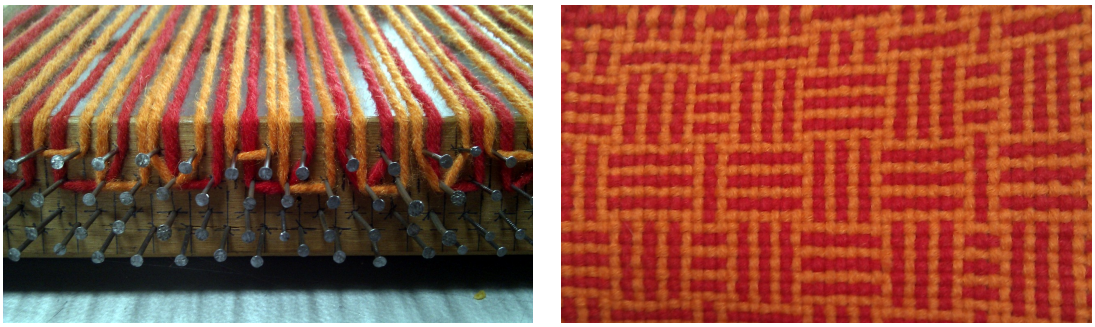
```
O:O:O:O:
```

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 text-based representation is too simple to really 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.



(a) The warped frame loom.

(b) Close-up of resulting weave.

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

There were two motivations behind this approach, firstly to begin to understand weaving by modelling plain weave, and confirming a hypothesis (the text version of the pattern) by following instructions produced by the language by actually weaving 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, although given the range of patterns possible, this was not an issue for this workshop. However the abstract nature of the symbolic modelling was more of a problem, in future developments we addressed both of these issues.

3 Four shaft loom simulation

Our four shaft loom simulation takes a different approach to that of most weaving simulation software. One of our core aims is to gain knowledge about the computational processes involved in weaving, and so need to gain deep understanding of how a loom works. 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 almost the exact inverse - you describe the set up of the loom first, and it 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, only considering the visual result in terms of the underlying structure 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 – an 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 provides 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. Here follow some example weaves. Colour wise, in all these examples the order is fixed – both the warp and the weft alternate light/dark yarns.

As with testing our understanding of plain weave as described in the previous section, our next step was to try weaving the structures with a real loom and threads, in order to test the patterns produced were correct. A frame loom was constructed to weave these patterns, shown 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 approach as that used in warp weighted looms, much faster than counting threads manually each time. It is important to use thinner threads than the warp for the heddles, but as they are put under tension during the weaving process they do need to be strong. Fittingly for this project, configuring the warp with heddles felt very much like coding threads, with threads.

Often a form of improvisation is required when weaving, even when using a predefined pattern. There is a lot of reasoning required in response to issues of structure that cannot be defined ahead of time. You need to respond to the interactions of the materials and the loom itself. The most obvious problem you need to think about and solve 'live' as you go, is the selvedge – the edges of the fabric. 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 which interacts with this.

In relation to computer programming, this improvisation at the loom is analogous to *live coding*, where code is written 'on the fly', often as a performing art such as music making. See Emma Cocker's article in the present issue of Textile for deep investigation into the relation

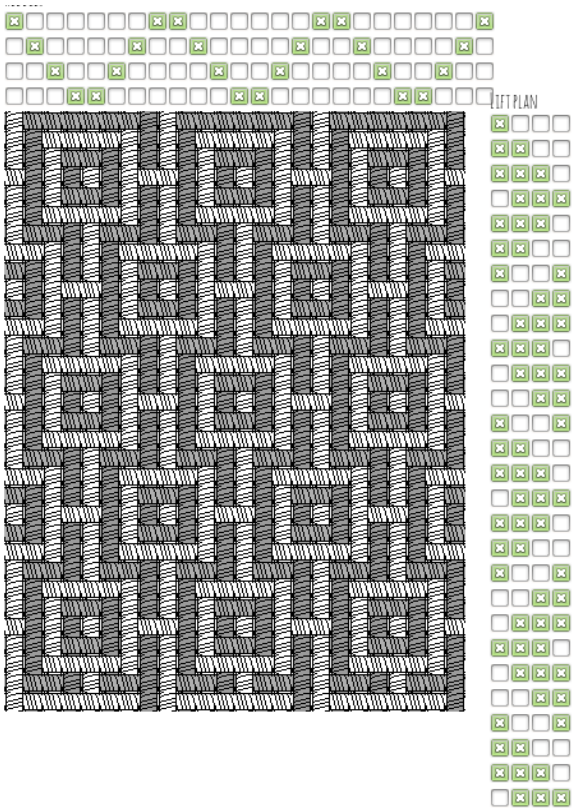


Figure 2: The interface for our four shaft loom simulation, showing heddles (above), lift plan (to the right) and simulated weave.

between between live weaving and live coding.

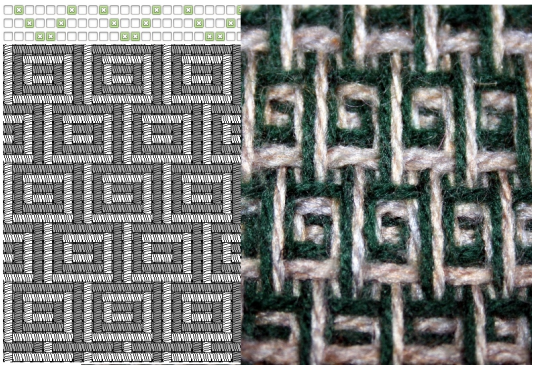
Figure 3b shows close-ups of the simulated *meander* pattern and its actual weave. There are clear differences visible between them, due to the behaviour of the long 'floating' threads; the pattern would be distorted further if the fabric were removed from the loom and the tension lost. The extent to 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

(a) Frame loom constructed to test our simulation

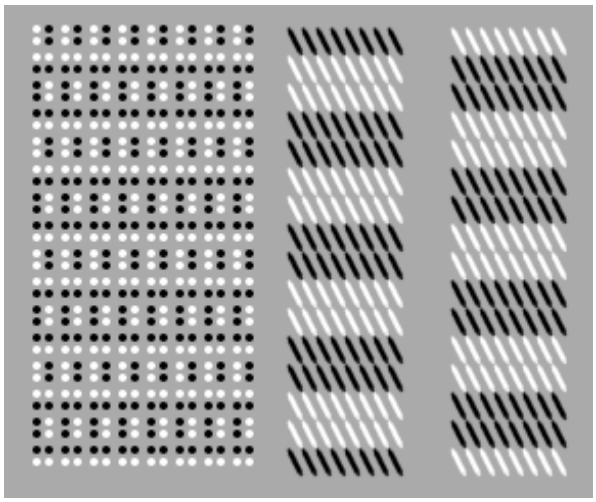


(b) Close-ups of a simulated *meander* pattern, and its actual weave (with floating threads)

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

produces long strips of fabric, which in antiquity were used as the starting bands and that form the borders of a larger warp weighted weaving.

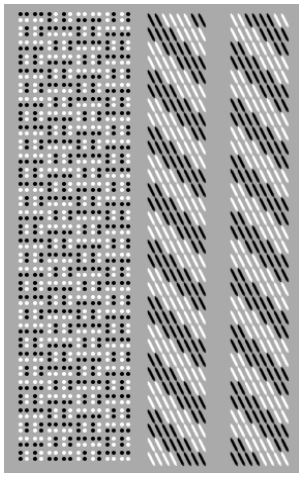
Tablet weaving is extremely complex, so in a similar manner to the 4 shaft loom, we devised a language/notation to help better understand it. As before, this language can be used either to drive a computer simulation, or can be followed when weaving. The following shows the output for a simple program written in this language, consisiting 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. `(weave-forward 16)` turns all the cards a quarter turn, adds one weft and repeats this 16 times.

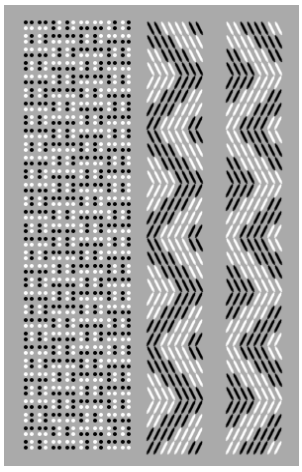
In our simulation, the cards are set up with a double face weave on square cards: black, black, white, white clockwise from the top right corner. We can offset these cards from each

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:



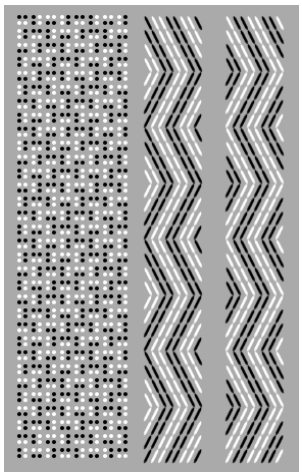
```
(rotate-forward 0 1 2 3 4 5)
(rotate-forward 0 1 2 3)
(rotate-forward 0 1)
(weave-forward 32)
```

One interesting limitation of tablet weaving is that it is not possible to weave 32 forward quarter rotations without completely twisting up the warp, so we need to go forward and backwards to make something physically weavable. However as the below demonstrates, if we do so, then a 'zig zag' pattern results.



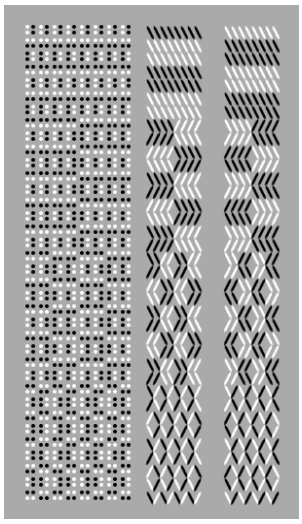
```
(rotate-forward 0 1 2 3 4 5)
(rotate-forward 0 1 2 3)
(rotate-forward 0 1)
(repeat 4
  (weave-forward 4)
  (weave-back 4))
```

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



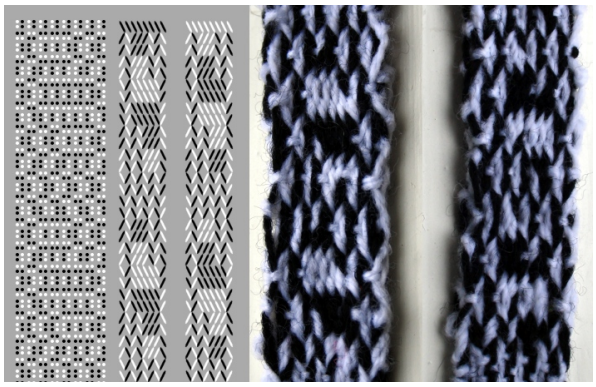
```
    (rotate-forward 0 1 2 3 4 5 6)
    (rotate-forward 0 1 2 3 4 5)
    (rotate-forward 0 1 2 3 4)
    (rotate-forward 0 1 2 3)
    (rotate-forward 0 1 2)
    (rotate-forward 0 1)
    (rotate-forward 0)
    (repeat 4
      (weave-forward 4)
      (weave-back 4))
```

As an alternative to specifying rotation offsets as above, we can use *twist* to form patterns. Accordingly, the `twist` command takes a list of cards to twist, and results in these cards effectively reversing their turn direction compared to the others, as demonstrated below. 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 with the woven form below, 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 language was the first we created that describes the actions and movement of the weaver. It was mainly of use in understanding the complexities of tablet weaving, indeed some of this 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 started investigations into combining the tablet and warp weighted weaving techniques into a single notation system. This remains a challenge, but pointed in the direction of a more general approach being required - rather than either a loom or weaver-centric view.

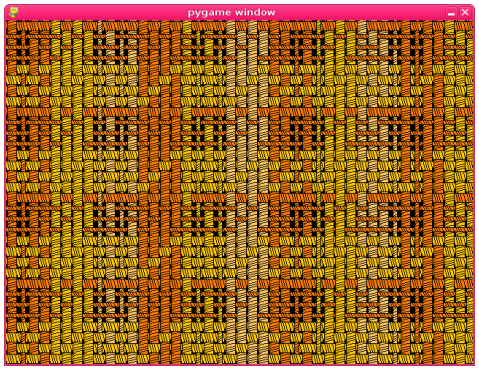
5 Flotsam Raspberry Pi Simulation

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, 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 used for the first mathematickal arts workshop, and describes weave structure and patterns with 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 different weave patterns than those possible with Jacquard looms, through the inclusion of additional yarn properties beyond colour. 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.

(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

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 @Horn2009, 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 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,

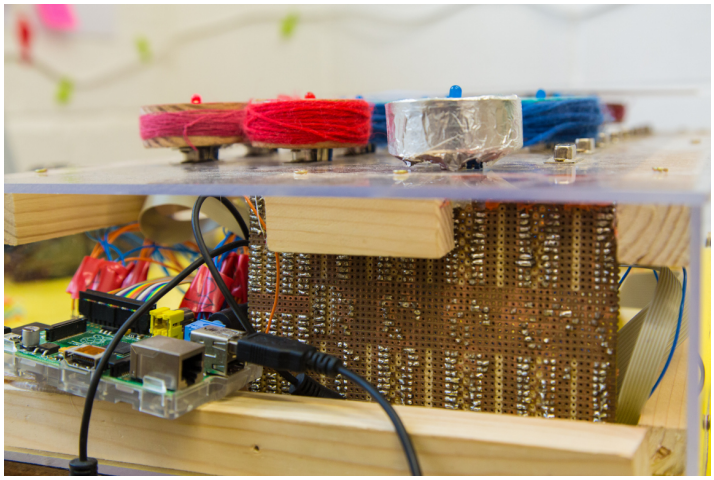
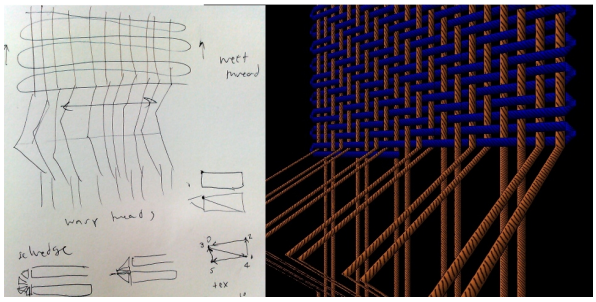
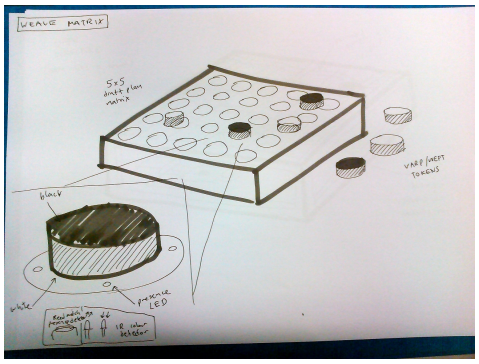


Figure 5: Profile of the Flotsam prototype, with tactile blocks in situ.



(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*

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 important constraint, partly due to the need for use in public places. We therefore designed the blocks as disks with no physical connections, painted white and black on different sides, and containing magnets so that the orientation and position of the disks, via *hall effect* sensors in the base. The hall effect sensors detect the polarity of nearby 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

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

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

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.

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 thread within code. A naive approach would be to represent it 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 the affordances of a programming language very well, which is well suited to the processing of lists. But this is only one point of view, and this grid-like view of a weave does not consider the path of a thread, including at the selvedge. What happens if we attempt to take the point of view of a thread?

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.

```
-- 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 (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
```

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* a 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 important part of the structure of a weave, at least while it is on a 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 last 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. Because a thread itself continually twists relative to the weave, it has little purchase on a left/right orientation. The concept of *in* and *out* operates relative to what has been woven before.

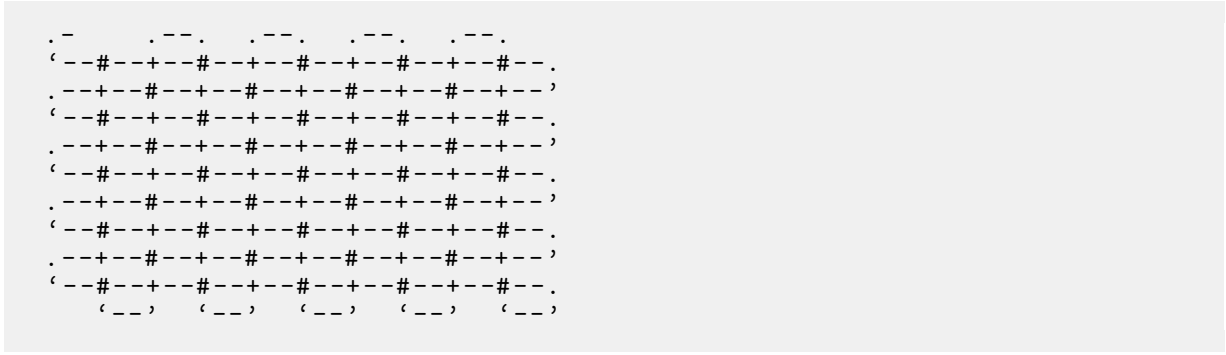
The concept of *in* and *out* fits well with the back-and-forth structure of a weave. Rather than thinking in terms of turning left on one side and right on the other, we can think of both turns in terms of the thread first turning *out* from the weave, and then back *in* to turn back on itself on the next row. Accordingly the creation of a warp of `n` threads and `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:

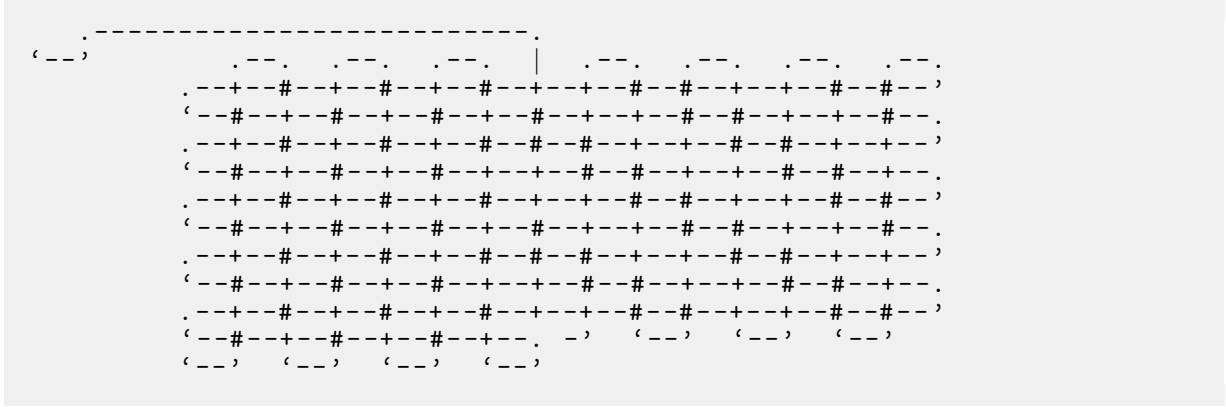
```
tabby h w = warp h w ++ [TurnIn]
                     ++ threadWeftBy (rot 1) ([Over,Under]) h w
```

Following this weave produces the following:



An advantage of this approach is that it is able to represent a case where the weft threads of a weave are pulled, in order to become a warp for a later stage of the weave. This 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 on a warp-weighted loom. 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 Dyadic arithmetic in the book of elements

Discrete mathematics is often now thought about in the technological context of modern computers, and we struggle to understand the connection with weaving. However, if we look back, we realise that the birth of discrete mathematics took place when weaving was the predominant technology. The hypothesis of Ellen Harlizius-Klck, which lies at the core of the present special issue, is that discrete mathematics in some sense *began* with thought processes

of weavers, and that this is implicit in the system of counting defined and applied in Euclid's Elements. To engage with this hypothesis, we here attempt to implement this system of counting in a contemporary programming language.

We will make this attempt using the Haskell programming language, known for its very strong focus on *types*. We will approach the definitions given in book VII of Euclid's Elements in turn, in the following, to see how far we can get, and what problems arise.

8.1 Definition 1: A unit is that by virtue of which each of the things that exist is called one.

```
data Unit = Unit
```

The first definition appears straightforward, but allows us to introduce our first piece of Haskell code, which simply defines a *data type*. On the left is given the name of the data type, in this case `Unit`, and on the right all the possible instances of that type, which here is again called `Unit`.

However, the above already has an error, in that in ancient Greece, a unit would not be thought of as abstract in this way; you could think about a sheep unit, or a cow unit, but ancient Greeks would find it nonsensical to think of a unit as being independent of such a category. Haskell allows us to model this as this by adding a parameter `a` for the type:

```
data Unit a = forall a. (Unit a)
```

As the `forall` suggests, the type `a` can represent any other type that we might define, such as `Sheep` or `Cow`. In practice this type parameter does nothing, apart from indicate that a `Unit` is thought about with reference to a concrete type of thing. So to model an unit of white Sheep, we could do the following:

```
data Sheep = Sheep {colour :: String}

sheep = Unit (Sheep "white")
```

However we are not interested in `Unit`s having a particular identity here, so we will use a definition which specifies a type parameter, but does not require a value when a Unit instance is being created:

```
data Unit a = forall a. Unit

sheep :: Unit Sheep
sheep = Unit
```

Our sheep here still has the type of `Unit Sheep`, but does not define anything about a particular sheep.

8.2 Definition 2: A number is a multitude composed of units.

The second definition also appears straightforward; in code terms, we can think of a *multitude* as a list, which is denoted by putting the `Unit` datatype in square brackets:

```
type Multitude a = [Unit a]
```

Note that the same type parameter is used in `Unit a` and `Multitude a`, which means that you for example can't mix `Sheep` and `Cows` in the same multitude.

However, there is awkward detail at play. Firstly, in Haskell it is valid to have an empty list, whereas the number *zero* is not defined by Euclid. Indeed, neither is the number *one* – a multitude begins with *two*, the number *one* is not considered to be a number, but simply a unit. A representation that truly represents a multitude should neither admit the numbers *zero* or *one*.

A way around this is to define a multitude relative to the number two, as a pair, or *dyad* of units:

```
data Multitude a = Dyad (Unit a) (Unit a)
    | Next (Unit a) (Multitude a)
```

The number *four* would then be constructed with the following:

```
four = Next Unit (Next Unit (Dyad Unit Unit))
```

It would be nice if we could 'see' this multitude more clearly. We can visualise it by telling Haskell to show a `Unit` with an `x` and by stringing together the units across instances of `Dyad` and `Next`:

```
instance Show (Unit a) where
  show x = "x"

instance Show (Multitude a) where
  show (Dyad u u') = show u ++ show u'
  show (Next u n) = show u ++ show n
```

Then the number `four` is shown like this (here the > prefixes the expression `four`, and the result is shown in the following line):

```
> four
xxxx
```

Here is a handy function for turning the integer numbers that we are familiar with into multitudes, which first defines the conversion from `2`, and then the general case, based upon that.

```
fromInt 2 = Dyad Unit Unit
fromInt n | n < 2 = error "There are no multitudes < 2"
          | otherwise = Next Unit $ fromInt (n-1)
```

8.3 Definition 3. A number is a part of a number, the less of the greater, when it measures the greater

Definition 3 is a little more complex, and because a single unit is not a number, awkward to express in contemporary programming languages. Lets begin by defining the cases where `lesser` is true, by following two multitudes, and returning `True` if we get to the end of the left hand multitude first. We may also define `greater` by simple swapping the two parameters:

```
lesser (Dyad _ _) (Next _ _) = True
lesser (Next _ a) (Next _ b) = lesser a b
lesser _ _ = False

greater a b = lesser b a
```

We may define the equality operator == by taking the same approach:

```
instance Eq (Multitude a) where
  (==) (Dyad _ _) (Dyad _ _) = True
  (==) (Next _ a) (Next _ b) = a == b
  (==) _ _ = False
```

We can then define a function `isPart` in terms of `lesser` and another function `measures` which returns True if it is able to repeatedly subtract one multitude from another (using `measureAgainst`), until they are equal.

```
measureAgainst :: Multitude a → Multitude a → Maybe (Multitude a)
measureAgainst (Next _ x) (Next _ y) = measureAgainst x y
measureAgainst (Dyad _ _) (Next _ (Next _ x)) = Just x
measureAgainst _ _ = Nothing

measures :: Multitude a → Multitude a → Bool
measures a b | a == b = True
             | isJust remaining = measures a (fromJust remaining)
             | otherwise = False
  where remaining = (measureAgainst a b)

isPart :: Multitude a → Multitude a → Bool
isPart a b = lesser a b && measures a b
```

8.4 Definition 4. But parts when it does not measure it.

This is now straightforward to represent:

```
isParts :: Multitude a → Multitude a → Bool
isParts a b = lesser a b && not (measures a b)
```

The usefulness of the definition of *parts* comes in later definitions, particularly where ratios are dealt with.

8.5 Definition 5. The greater number is a multiple of the less when it is measured by the less.

Again, straightforward to represent in terms of what we have defined already:

```
isMultiple :: Multitude a → Multitude a → Bool
isMultiple a b = greater a b && not (measures a b)
```

8.6 Definition 6. An even number is that which is divisible into two equal parts.

This definition of even seems straightforward, but from a contemporary perspective, is made awkward by the exclusion of the number *one* from the class of numbers. In particular, the number two cannot be divided into two parts, if we assume from definition 3 that a part must be a multitude. The approach we take below then is to attempt to divide a number into equal parts by first taking two units from it, and then further successive units until either we have taken a number equal to what we are left with, in which case the number is even, or greater than what we are left with, in which case it is not. This excludes the number `2` from the class of even numbers, but nonetheless is true to Euclid's definition, at least according to the English translation we are using.

```
removeUnit (Dyad _ _) = error "Cannot remove a unit from two"
removeUnit (Next _ n) = n

addUnit n = Next Unit n

balance :: Multitude a → Multitude a → Maybe (Multitude a)
balance a b | lesser a b = Nothing
            | a == (Dyad _ _) = Nothing
            | a == b = Just (a, b)
            | otherwise = balance (removeUnit a) (addUnit b)

isEven (Next u (Next u' a)) = isJust (balance a (Dyad u u'))
isEven _ = False
```

8.7 Definition 7. An odd number is that which is not divisible into two equal parts, or that which differs by a unit from an even number.

The definition of odd numbers is in two parts. It is debatable whether the number `3` meets the first part; in a sense, `3` is not divisible into parts at all, as a unit is not a multitude. However it is clear that it does meet the second part, and so according to boolean logic, this amiguity is removed.

```
isOdd a = partOne a || partTwo a
  where partOne (Next u (Next u' a)) = (isNothing (balance a (Dyad u u')))
        partOne _ = False
        partTwo a = isEven (addUnit a)
```

8.8 Definition 8. An even-times-even number is that which is measured by an even number according to an even number.

We first need to find the number of times that a number measures another number. This is done by the following function `measure`, which is a little awkward to define without the number one.

```
measure :: Multitude a → Multitude a → Maybe (Multitude a)
measure a b = do m ← measureAgainst a b
                 if m == a
                   then (return two)
```

```
                    else do m' ← measureAgainst a m
                             measure' two a m'
  where measure' m a b | a == b = Just (addUnit m)
                       | otherwise = do b' ← measureAgainst a b
                                        measure' (addUnit m) a b'
```

We can then find the even-times-even numbers using an exhaustive search:

```
isFTimesF f f' m = or $ catMaybes [(f <$> measure a m)
                                   | a ← filter f' (lessers m)]

isEvenTimesEven = isFTimesF isEven isEven
isEvenTimesOdd = isFTimesF isEven isOdd
isOddTimesEven = isFTimesF isOdd isEven
```

We have also defined odd-times-even and even-times-odd numbers, which are definitions 9 and 10.

8.9 Dyadic mistakes

Having come part of the way to implementing Dyadic arithmetic in the Haskell type system, we have undoubtedly made mistakes. In particular, it would seem that while units are not considered to be multitudes, avoiding using them in calculations altogether is not teneble and leads to errors. The relationship between parts, numbers and units is perhaps not straightforward to model with a rigid type structure.