Textility of Code: A catalogue of errors

Introduction

<> (removed - this seems beside the point.. could go in intro of the journal issue instead. In the following we explore the notion of code as weave. A challenge for any discussion of coding of weaves is the Jacquard mechanism in machine looms, often recounted as the first meeting point of digital technology and weaving. This is wrong for many reasons, but most crucially, weaving involves the interaction of discrete threads and so has itself always been digital technology since prehistoric times. Furthermore, code involves human enagement with structure, whereas the Jacquard mechanism allows weaves to be considered as two dimensional images, rather than the three dimensional structures which give rise to them.)

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.

Citation test [**?**]

Dyadic arithmetic in the book of elements

Now discrete mathematics is 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 Klück (TODO: cross ref Ellen's article), which lies at the core of the present special issue, is that discrete mathematics 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 attempted 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 defining the types of things in clear way. We will approach the definitions given in book VII in turn, in the following, to see how far we can get, and what problems arise.

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 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.

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

The second definition again appears straightforward; in code terms we can think of a multitude as a list, which is denoted by putting the `Unit` datatype in 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 again, the detail makes this a little bit complex. Firstly, such a list is allowed to be empty, whereas the number zero is not represented by this definition. Indeed, neither is the number one – a multitude begins with two, less than that is not considered to be a number.

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

```haskell
data Multitude a = Pair (Unit a) (Unit a)
    | Next (Unit a) (Multitude a)
```

The number four would then be constructed with the following:

```haskell
four = Next Unit (Next Unit (Pair 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 `Pair` and `Next`:

~~{.haskell .colourtex} instance Show (Unit a) where show x = "x"

instance Show (Multitude a) where show (Pair u u') = show u ++ show u' show (Next u n) = show u ++ show n ~~{.haskell .colourtex}

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

```
> four
xxxx
```

Here is a handy function for turning integers into multitudes, which first defines the conversion from 2, and then the general case, based upon that.

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

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, or otherwise false:

```haskell
lesser (Pair _ _) (Next _ _) = True
lesser (Next _ a) (Next _ b) = lesser a b
lesser _ _ = False
```

```
greater a b = lesser b a
```

We can define the equality operator == with the same approach:

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

Toothpaste

The 'toothpaste' approach to weave simulation was the result of both working with the 3D selvedge calculation with the warp weighted loom model, and wanting a more general approach to modelling that could eventually be expanded to include the double twining of tablet weaving. This will enable us to finally represent the ancient technique of combining different weaving technologies into one fabric.

It is designed to be driven by our single thread notation system, which frees us from concepts of the loom, weaver or even warp and weft.

This example is mix of tabby and 2:2 twill, created by this code:

```
warp 12 24 ++ [TurnIn] ++ threadWeftBy'' Odd (rot 3) ([Over,Under]) 12 12 ++ threadWeftBy''
```

This line of code produces a large list of instructions the weave renderer uses to build the model, turning the thread and shifting it up and down as it crosses itself.

Here is a different view of the same fabric to show the selvedge:

We can also now easily introduce other changes to the yarn structure, for example modifying the width along the length of the thread.

tablet weaving simulation

Tablet weaving is an ancient form of pattern production using cards which are rotated to provide different sheds between warp threads. It's used to produce long strips of fabric, or the starting bands and borders that form part of a larger warp weighted weaving.

Tablet weaving is extremely complex, so we devised a language/notation for understanding it better. This language can be used either to drive a simulation, or can be followed when weaving.

The language consists of simple instructions, for example:

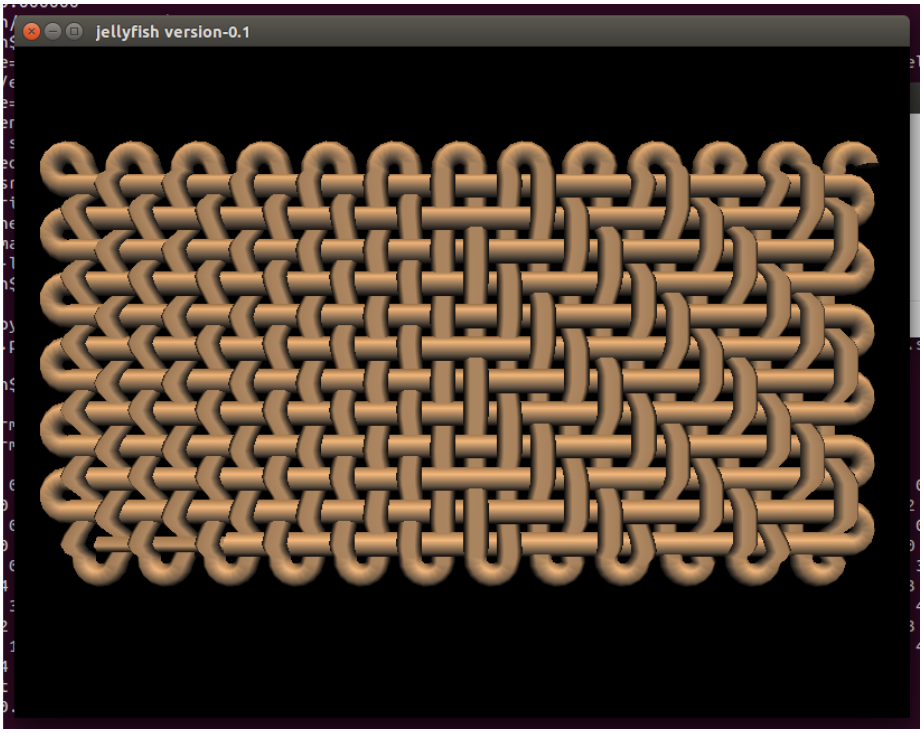

Figure 1:

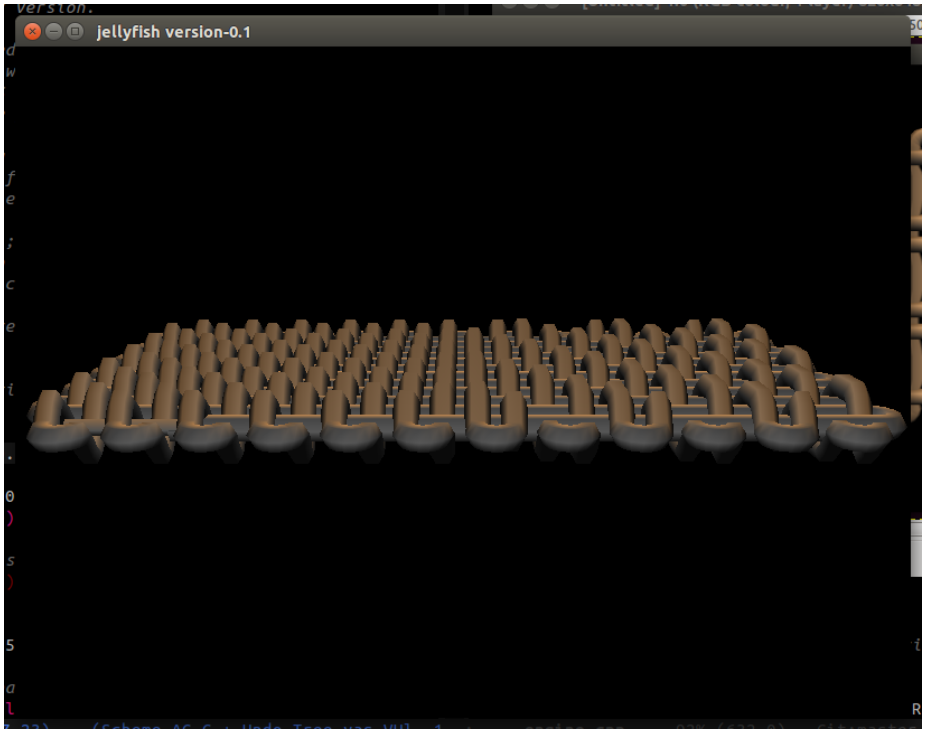

Figure 2:

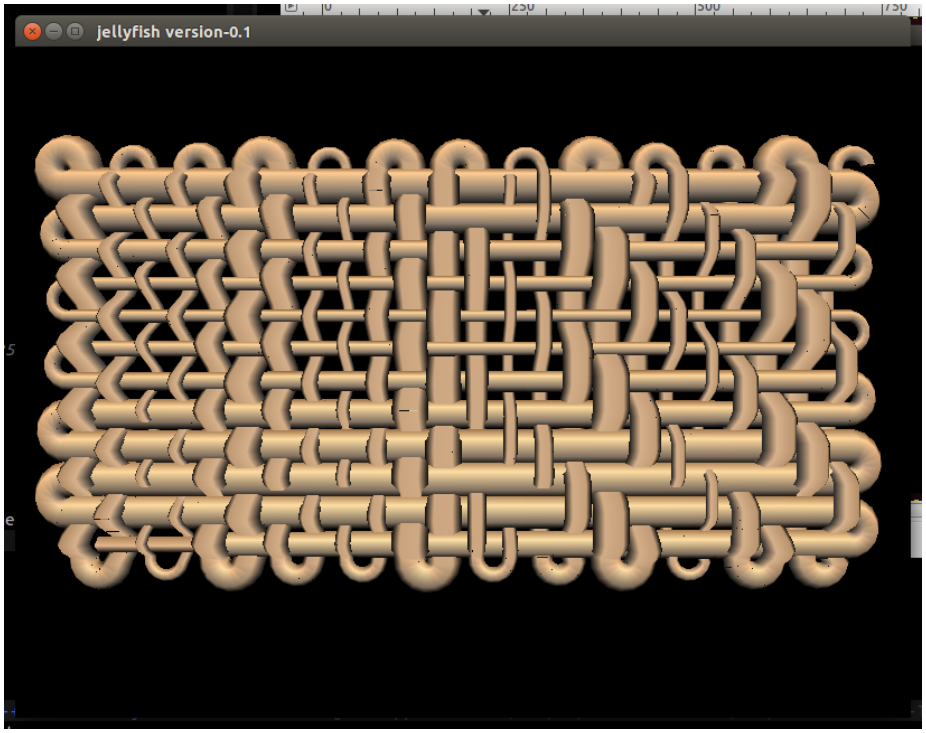

Figure 3:

```
(weave-forward 16)
```

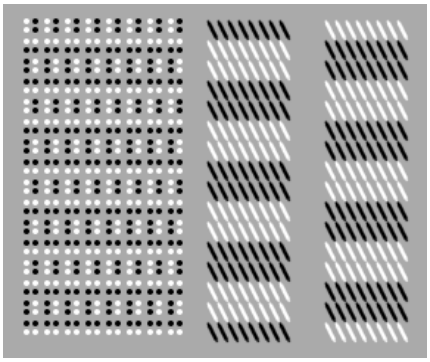

Figure 4:

The card rotations are shown on the left for each of the 8 cards in this case, the predicted weaving is on the right for the top and bottom of the fabric. This is setup with a double face weaving on square cards, so black, black, white, white in clockwise from the top right corner. `(weave-forward 16)` turns all the cards a quarter turn, adds one weft and repeats this 16 times.

We can offset the cards from each other first to make a pattern. `rotate-forward` turns only the specified cards a quarter turn forward without weaving a weft (`rotate-back` also works):

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

One interesting limiation of tablet weaving is that We can't really weave 32 forward quarter rotates without completely twisting up the warp so we need to go forward/back 8 instead to make something physically weavable:

```
(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))
```

Now we get a zigzag – if we change the starting pattern again:

```
(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)
```

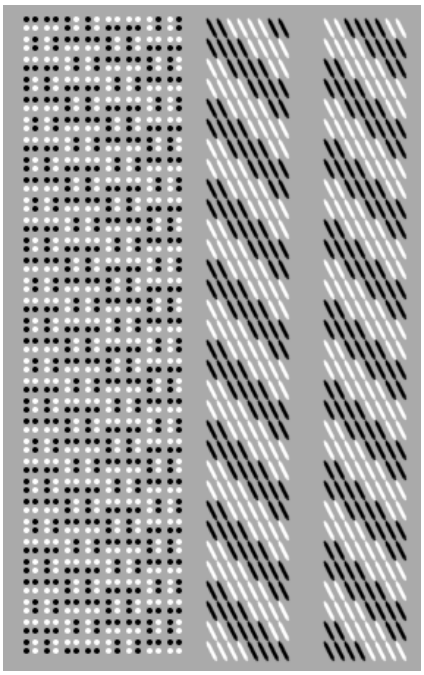

Figure 5:

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

This zigzag matches the stitch direction better. Instead of the rotation offsets we can also use twist to form other patterns. The `twist` command takes a list of cards to twist, and results in these cards effectively reversing their turn direction compared to the others.

```
(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)
```

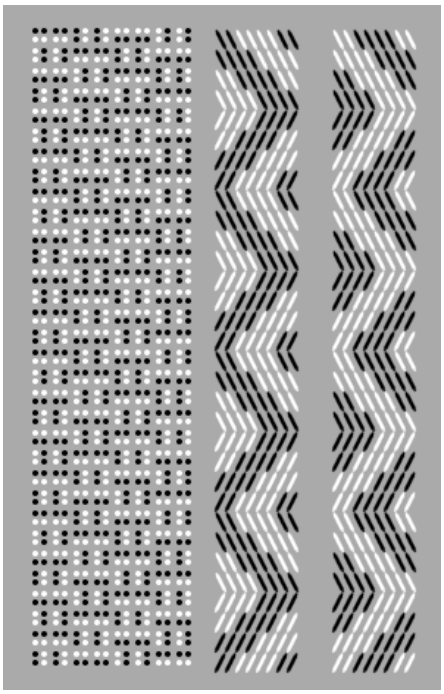

Figure 6:

Figure 7:

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

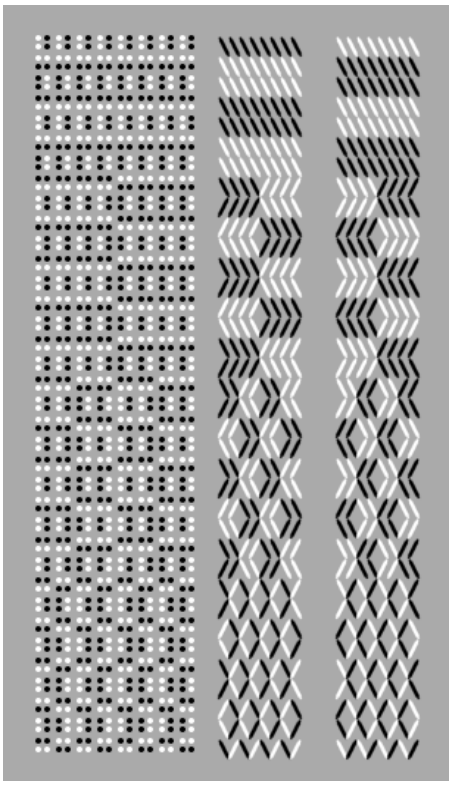

Figure 8:

With double faces weave, the twist needs to happen when the cards are in the right rotation – if we repeat this example, but change the first `(weave-forward 7)` to `(weave-forward 6)` we get this instead:

If we put the twists in the loops, we can make small programs with complex results. You can see a comparison with the woven form below, this was created by following the program manually.

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

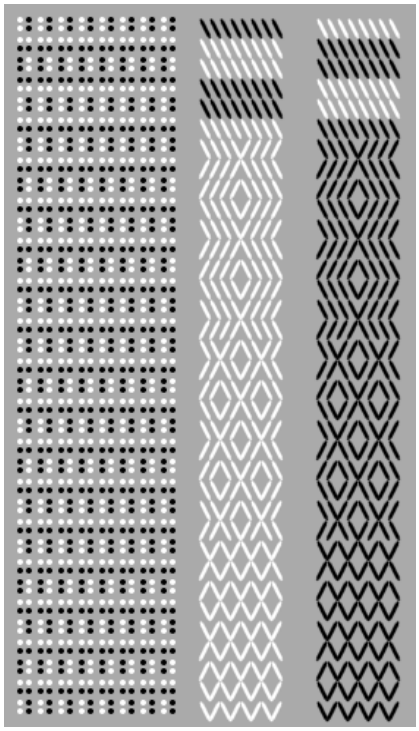

Figure 9:

```
(weave-forward 4)
(twist 5)
(weave-back 4))
```

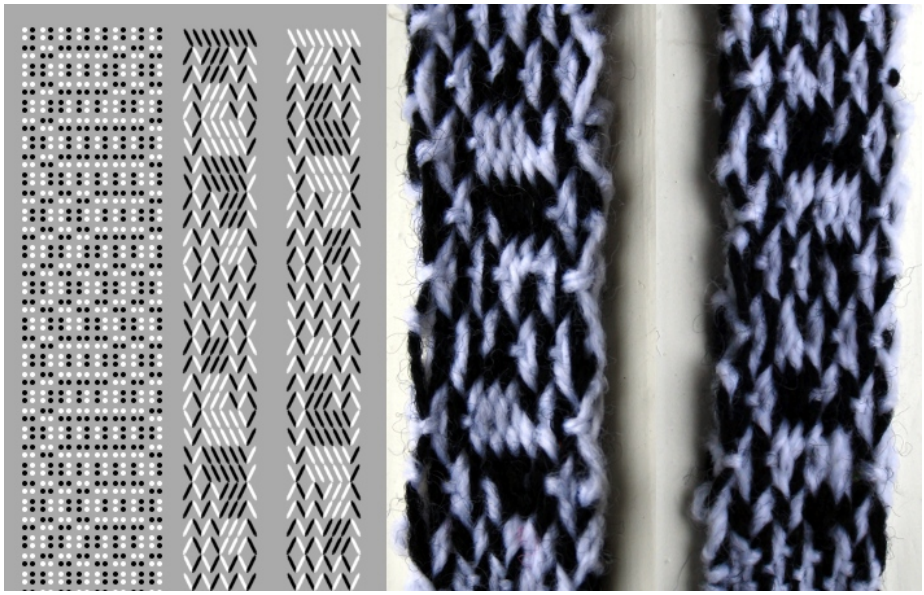

Figure 10:

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, problably due to the double twining of the weave. In some cases it has very different results, in others it matches perfectly.

This langauge 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 eigther a loom centred or weaver centred view.

Flotsam Raspberry Pi Simulation

Flotsam is a prototype screenless tangible programming language largely built from driftwood. Constructed in order to experiment with tangible hardware for teaching children programming, it is based on the same L system as used for the mathematickal arts workshop. It describes weave structure and patterns with wooden blocks representing yarn width and colour. The L system production rules for the warp/weft yarn sequences are constructed from the the positions

the blocks are plugged into using a custom hardware interface.

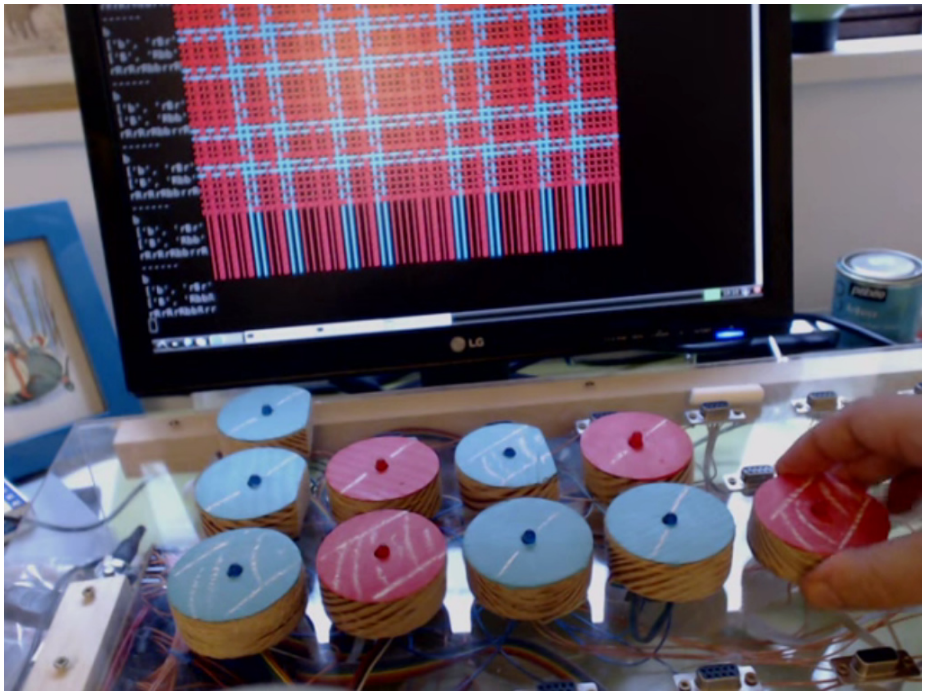

Figure 11:

The weaving simulation is running on a Raspberry Pi computer which is simultaneously reading the yarn token block hardware. The system is designed to describe different weave patterns than those possible with Jacquard looms, by including simple additional yarn properties beyond colour. The version shown in the figure above is restricted to plain weave, but more complex structures can be created as shown below:

The flotsam tangible hardware was used in primary schools and private tutoring with children, and was designed so the blocks could be used in many different ways. Experiments beyond weaving included an interface with the Minecraft 3D game and a live music synthesiser.

As in the Mathematickal Arts workshop, the L system approach is good for quick exploration of the huge variety of weaving patterns. However, one of the core goals for the weavingcodes project was to develop artefacts and interfaces for understanding how weavers think, and this was a markedly different approach - so proved challenging for this aim.

The visualisation technique also demonstrated an interesting limitation, as in common with many similar models does not take into account the selvedge of the fabric - it is conceptually an infinite grid of cellular elements rather than

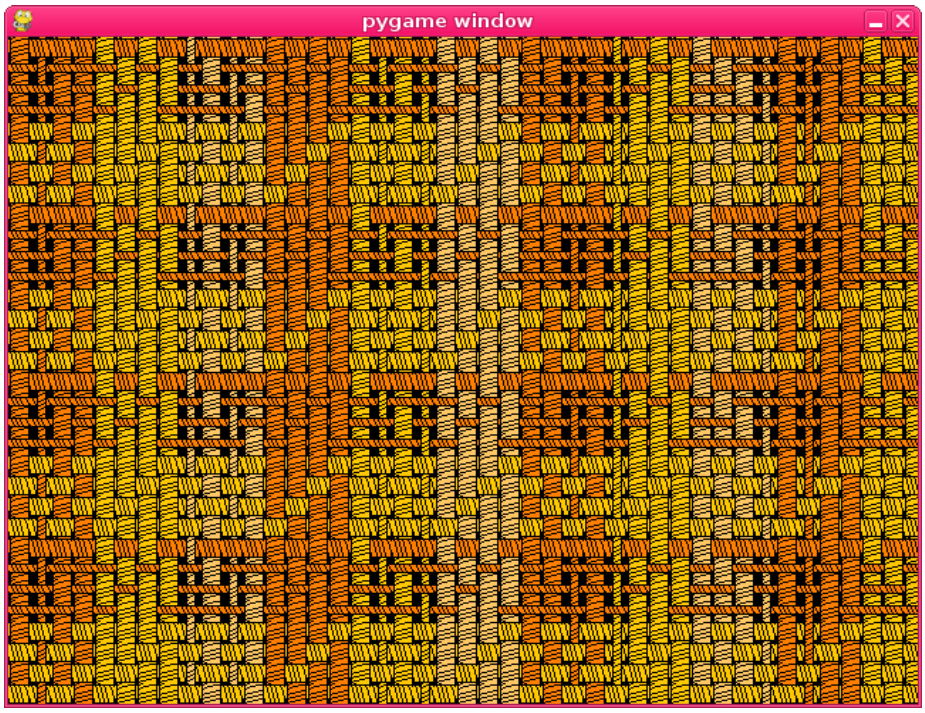

Figure 12:

representing a continuous thread.

During use we observered the findings of ["Comparing the Use of Tangible and Graphical Programming Languages for Informal Science Education": http://cci.drexel.edu/faculty/esolovey/papers/chi09.horn.pdf] in that the tangible interace proved better for collaborative learning than a wholly screen based system. The design of the system itself needed further development, which we followed up in the 'pattern matrix' - a tangible system designed specifically for weaving.

Four shaft loom simulation

The four shaft loom simulation inverts most weaving simulation techniques. Instead of defining the pattern you want directly, you describe the set up of a 4 shaft loom – so the warp threads that each of 4 shafts pick up in the top row of toggle boxes, then which shafts are picked up for each weft thread as the fabric is woven on the right (the lift plan).

This involved writing a model based closely on how the loom functions – for example calculating a shed (the gap between ordered warp thread) by folding over each shaft in turn and or-ing each warp thread to calculate which ones are picked up. This really turns out to be the core of the algorithm – here's a snippet:

```
;; '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 program is good for understanding how the loom setup corresponds to the patterns. Here are some example weaves. Colour wise, in all these examples the order is fixed – both the warp and the weft alternate light/dark yarns.

The next step was to try weaving the structures with real threads in order to

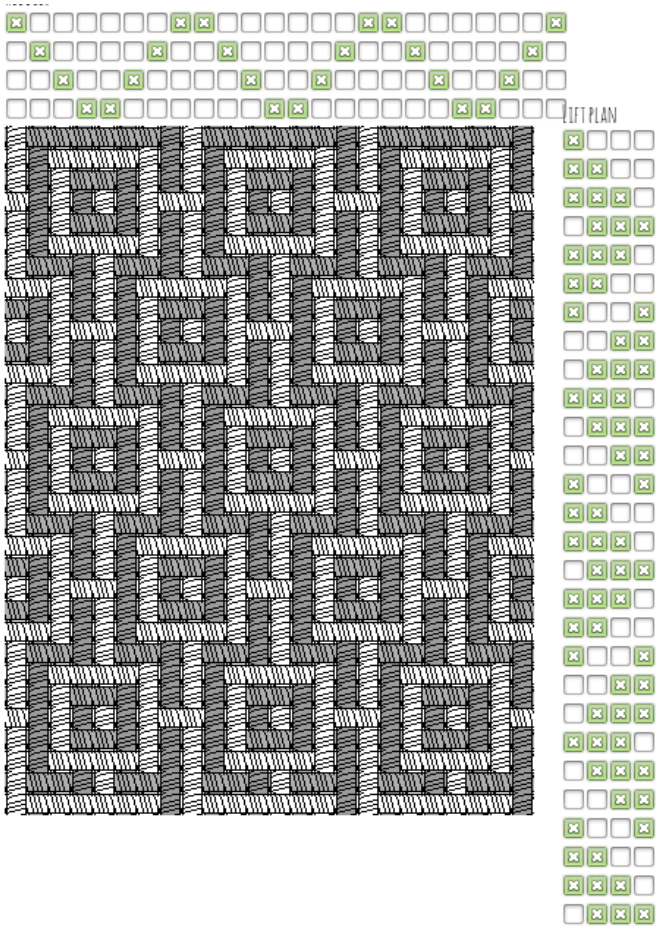

Figure 13:

test the patterns produced were correct. Ellen warned me that the meander weave would result in unstable fabric, but it would depend on the nature of the material used so was worth trying. A frame loom was constructed to weave these patterns.

Figure 14:

Here the shafts are sleyed to pick up the warp as defined by the simulation toggle buttons. 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 used in warp weighted looms and much faster than counting threads manually each time. It's important to use thinner threads than the warp, but you need to put quite a bit of tension on them so they need to be strong. There is something very appropriate in the context of this project about coding threads with threads in this way.

In relation to livecoding, 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' 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.

Figure 15:

Here's a closeup of the meander pattern compared to the simulation. The differences are due to the long float threads, which cause the pattern to distort further when the fabric is removed from the loom and the tension is gone. The extent to which it is possible or desirable to include these material limitations into a weaving language or model was one of our main topics of inquiry when talking to our advisors [Leslie], as well as the inability of even the highest range simulation software to do this fully.

In total there were three types of limitations noted. One is the selvedge, the lack of which in models and lanuages was mentioned earlier – another is floats as seen here.

The third is more subtle, some sequences of sheds cause problems when packing down the weft, for example if you are not too careful you can cause the ordering of the weft colours to be disrupted in some situations.

This was a step in the right direction, as the model represents weaving via the shed operation rather than a cellular matrix, it brings it closer to a continuous form.

Mathematickal arts workshop (foam brussels)

Plain or tabby weave is the simplest form of weaving, but when combined with sequences of colour it can produce many different types of pattern.

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 [citation needed] for use in hunting in Lowland Scotland.

The first weaving codes prototype I made was during the Mathematickal Arts workshop at FoAM Brussels [citation needed]. A few lines of Scheme calculate and print the colours of an arbitrarily sized plain weave, by using lists of warp and weft yarn as input.

```
; return warp or weft, dependant on the position
(define (stitch x y warp weft)
  (if (eq? (modulo x 2)
           (modulo y 2))
  warp weft))

; prints out a weaving
(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)))
```

We visualised the weaves with single characters representing colours for ascii text previewing, here are some examples:

Warp and weft all the same colour:

```
(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
```

2:2 alternating colour with an offset:

```
(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 looked quite promising as ascii art, but we didn't really know how it would translate into a textile. We also wanted to look into ways of generating new patterns algorithmically, using formal grammars. The idea is that you begin with an axiom, or starting state, and do a 'search replace' on it repeatedly following one or more simple rules:

```
Axiom: O
Rule 1: O => O : O :
Rule 2: : => : O :
```

So we begin with the axiom:

```
O
```

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

```
O : O :
```

Then run rule two, replacing `:` 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 :
```

The pattern grows like a cellular or plant structure - this technique was first developed for modelling plant growth.

We run the rules one more time, then read off the pattern replacing `O` for red and `:` as orange to warp a frame loom:

When weaving, we follow the same sequence for the weft threads:

The idea of this technique was twofold, firstly to begin to understand weaving by modelling plain weave, and confirming a hypothesis 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.

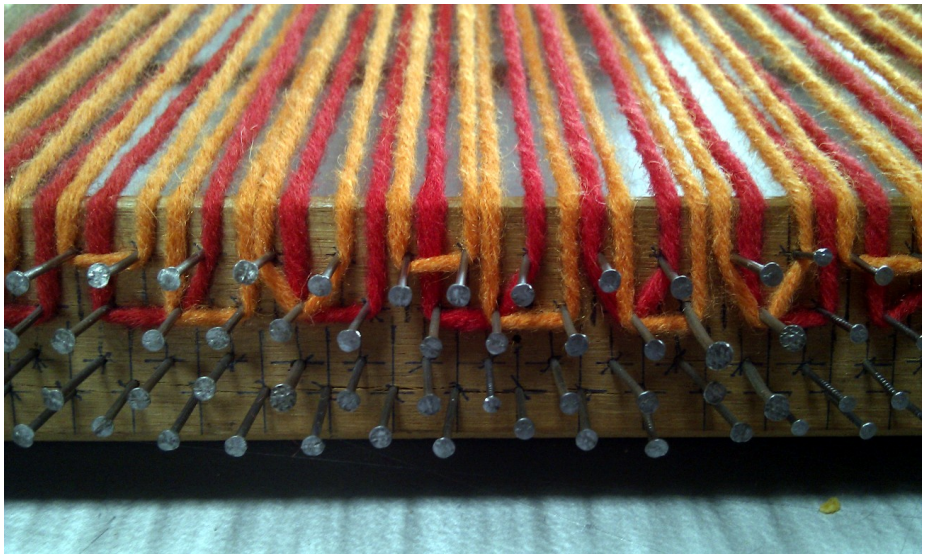

Figure 16:

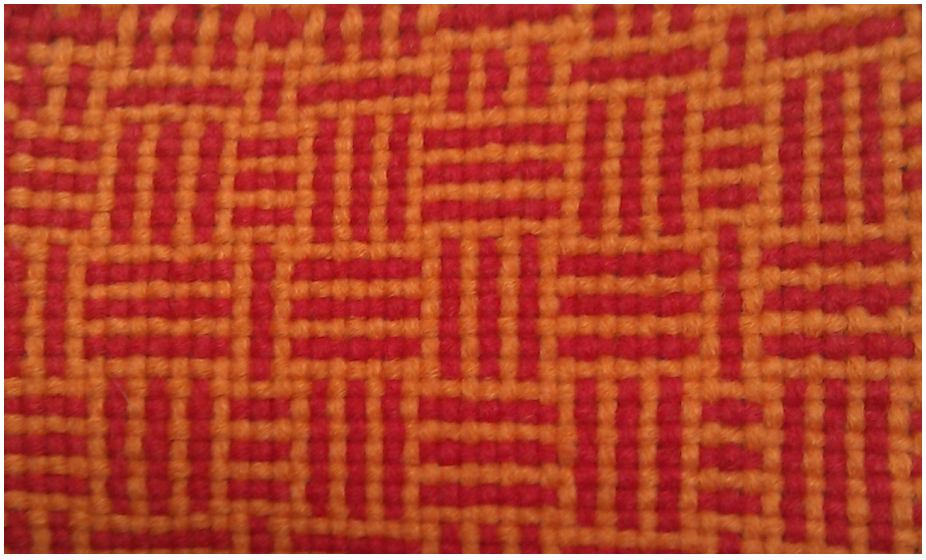

Figure 17:

This system was restricted by only working with plain weave, although given the range of patterns possible, this didn't seem too much of a problem. What was more problematic was the abstract nature of the symbols and text modelling, in future developments we addressed both of these issues.

Pattern matrix warp weighted loom simulation

One of the main objectives of the weavecoding project was to provide a simulation of the warp weighted loom to use in demonstrations and exploration of ancient weaving techniques. Beyond the previous weavecoding simulations we needed to show the actual process of weaving 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 that well.

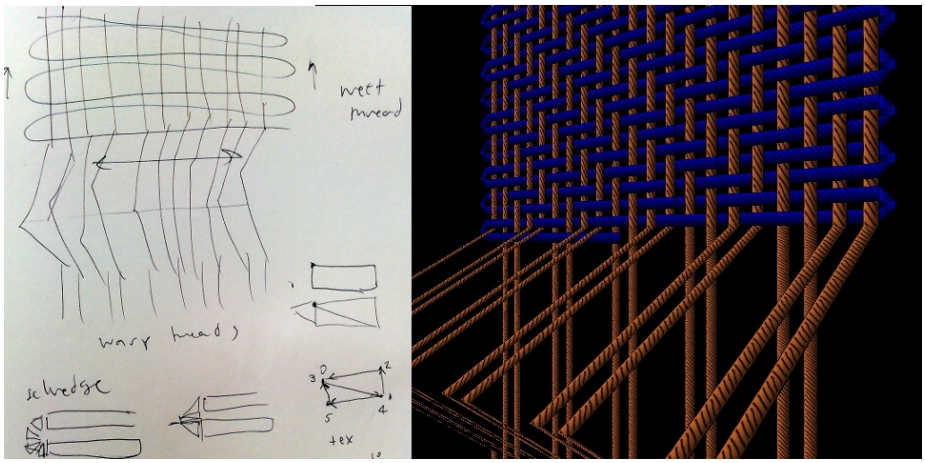

Figure 18:

We built a 3D procedural rendering system [jellyfish] to build the warp weighed loom simulation. This was developed specifically for the project, due to the needs of running on a Raspberry Pi computer, which allows for easy integration with our experimental hardware.

After out experience with Flotsam, we needed to explore tangible programming further. The pattern matrix was the next step, open hardware specialised for weaving and built by Makernow[] and FoAM Kernow for use in an extra care housing scheme alongside other crafts and technology workshops as part of Future Thinking for Social Living[http://ft4sl.tumblr.com/]. We also tested it in a museum setting during a residency at Munich's Museum für Abgüsse Klassischer Bildwerke (Museum of Casts of Classical Sculpture).

A primary objective of this prototype was to remove the need for physical plugs,

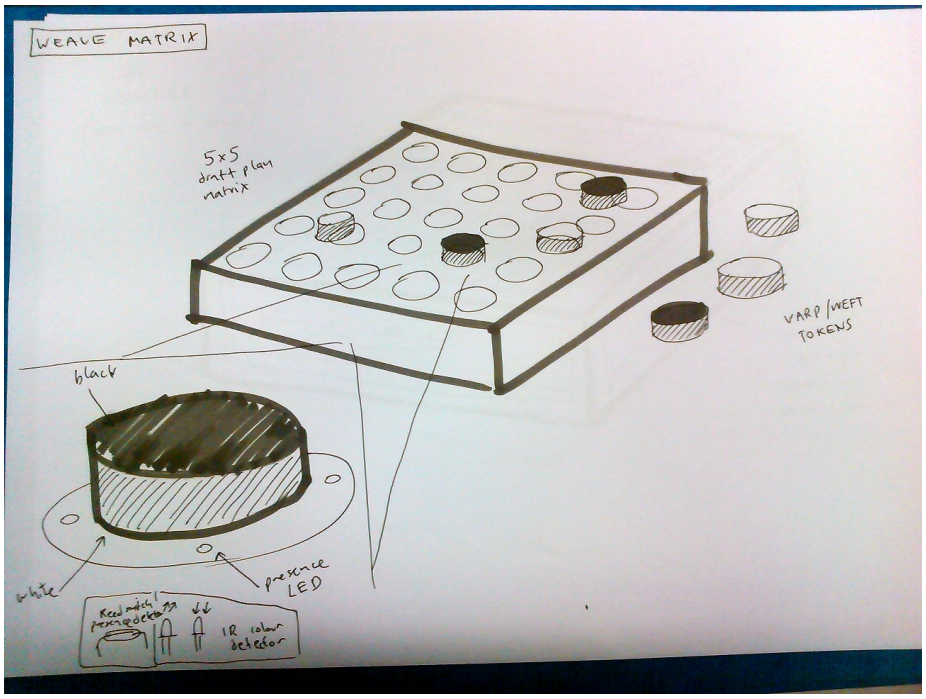
WEAVE MATRIX
5x5
draft plan
matrix
WARP/WEFT
TOKENS
black
white
pressure
LED
Reed switch
IR colour
detector

Figure 19:

which proved problematic as it took people time to learn how to align the blocks to plug them in to the Flotsam prototype. The cheapness and availabily of the programming blocks themselves was important to maintain (partly due to the need for use in public places) so we used blocks with no connections, painted white and black on different sides and detect their orientation and position via a magnet in the centre and hall effect sensors in the base.

Hall effect sensors allow us to detect the polarity of nearby magnetic fields – and seem to be restricted enough in range that they can be very precise. 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 1 bit of information to be detected per block, but for other features such as yarn colour selection we needed to be able to represent more information (4 bits were encoded in the flotsam blocks).

We used 4 hall effect sensors in a square which meant that we could detect rotation and flipping of the blocks. Incidentally, this gets very close to tablet weaving - in terms of the notation and the actions required to use the device. We can also represent all 16 possible states with only 4 blocks – if negative is 0 and positive is 1, and we read the code as binary clockwise from top left these are the four blocks changing their states with twist and flip:

Starting state [0,1,5,6]

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

Rotate clockwise [0,2,10,12]

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

Horizontal flip [15,11,10,12]

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

Rotate counter-clockwise [15,13,5,6]

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

Vertical flip [0,4,5,6]

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

Here was the design for the sensor PCB that contained the hall effect sensors under each programming block.

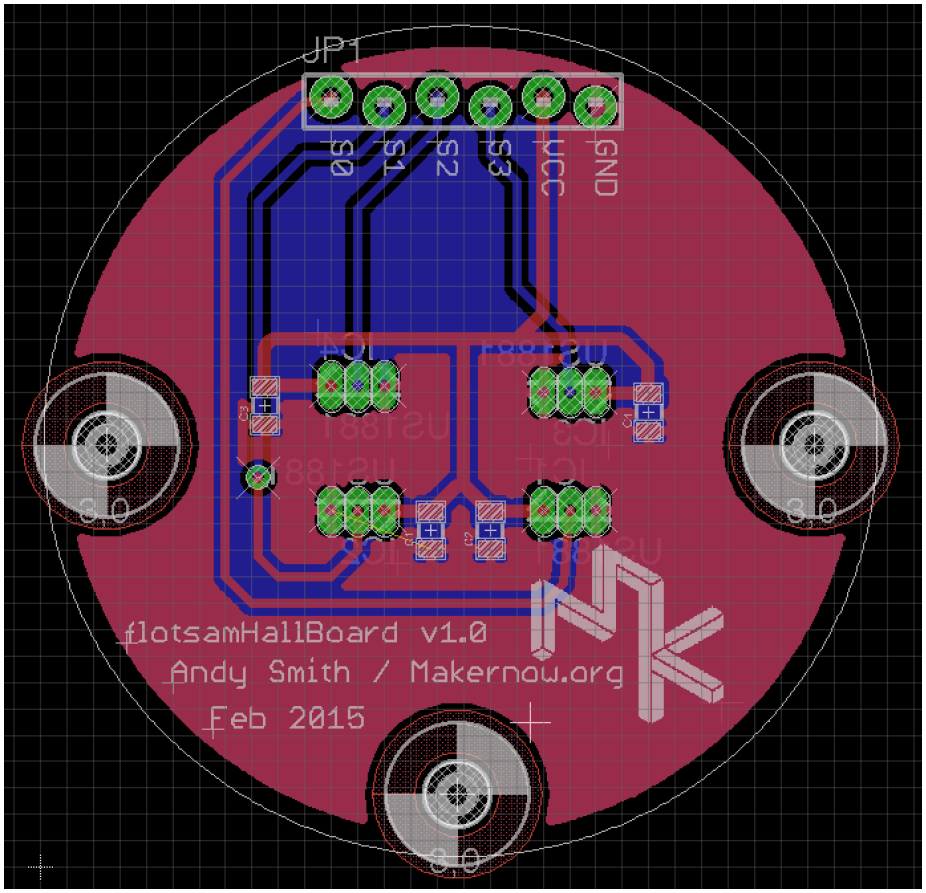
JP1
S0
S1
S2
S3
VCC
GND
flotsamHallBoard v1.0
Andy Smith / Makernow.org
Feb 2015

Figure 20:

A member of staff at Miners Court trying the first working version of the tangible weavecoding - in this case the Raspberry Pi displays the weave structure on the simulated warp weighed loom with a single colour each for warp and weft threads.

Figure 21:

As with the Flotsam prototype, we wanted to keep the system flexible in terms of later changes and modifications. The blocks are read by a AVR microcontroller per row which have a set of multiplexers each that allow you to choose between 20 sensor inputs all routed to an analogue input pin on the AVR. We're just using digital here for the hall effect magnet state, but it means we can try totally different combinations of sensors without changing the rest of the hardware.

After testing it with elderly people at our Miners Court residency there were a couple of issues. Firstly the magnets were really strong, and we worried about leaving it unattended with the programming blocks snapping together so violently. The other problem was that even with strong magnets, the placement of the blocks needed to be very precise. This is probably to do with the shape of the magnets, and the fact that the fields bend around them and reverse quite short distances from their edges.

To fix these bugs it was a fairly simple matter to take the blocks apart, remove 2 of the 3 magnets and add some rings to guide placement over the sensors properly:

The warp weighted loom simulation was the first one we designed that included selvedge calculation, as well as animating the shed lift and weft thread movement.

The threads are calculated in 3D but are represented by thin 'ribbons' with shading to make them look like they have a round cross section. This was due to speed limitations on the Raspberry Pi.

Figure 22:

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 didn'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 these complexities in this ancient weaving technique.