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 [**?**]

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 [http://fo.am/mathematickal_arts/] 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 1. When weaving, we followed the same sequence for the weft threads, resulting in the textile shown in Figure 2.

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

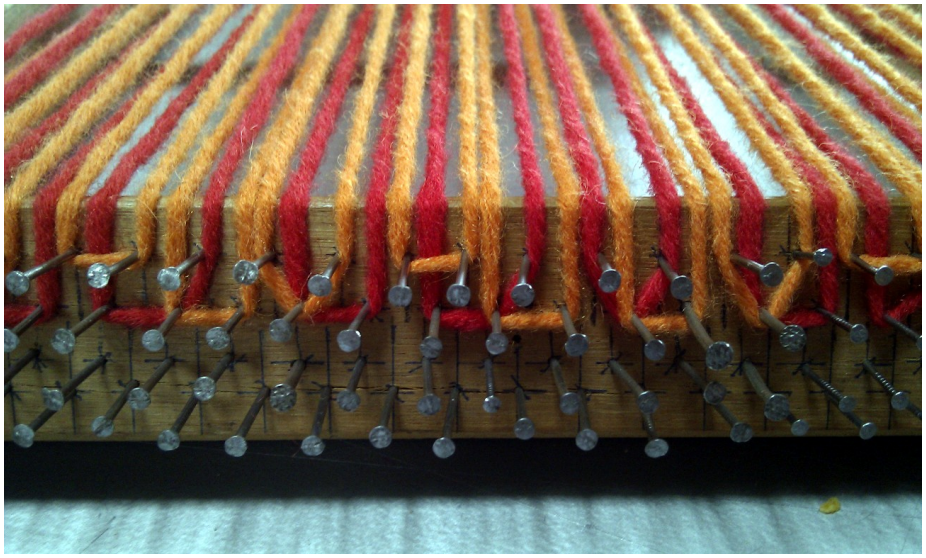

Figure 1: A warped frame loom, with colour pattern generated from an L-system.

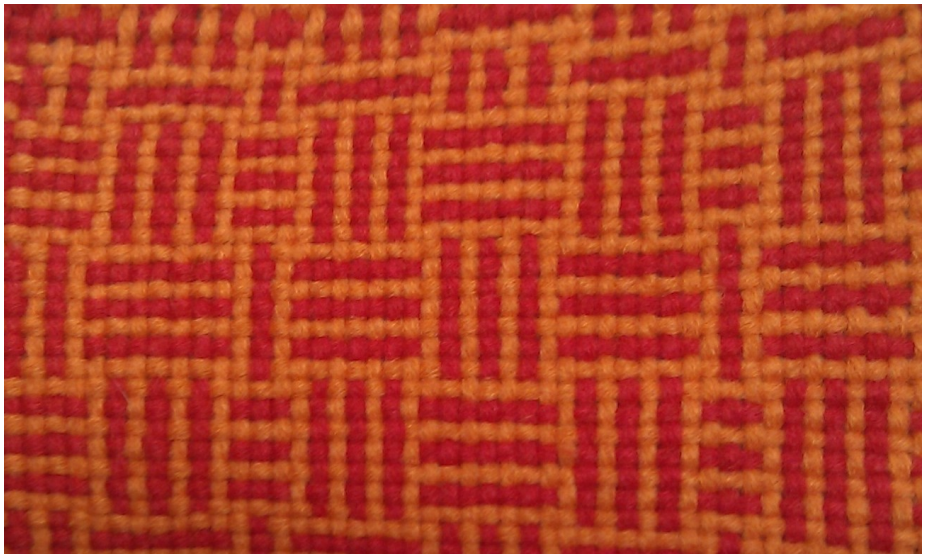

Figure 2: Close-up of weave resulting from L-System pattern.

the abstract nature of the symbolic modelling was more of a problem, in future developments we addressed both of these issues.

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.

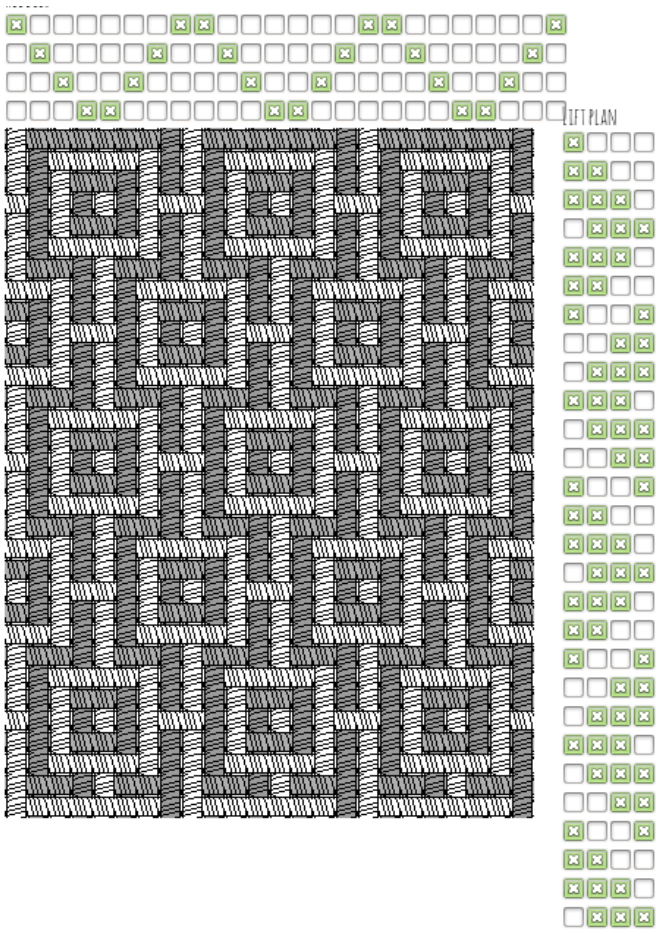

Figure 3: The interface for our four shaft loom simulation, showing heddles (above), lift plan (to the right) and simulated weave.

As with testing our understanding of plain weave 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 4. Here the shafts are sleyed to pick up the warp as defined by the simulation's input (the checkboxes) seen in Fig. 3). 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.

Figure 4: Frame loom constructed to test the four shaft loom simulation.

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 between between live weaving and live coding.

Figure 5: Close-ups of a simulated *meander* pattern, and its actual weave with floating threads.

Figure 5 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.

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 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. In the same way as before, this language can be used either to drive a computer simulation, or can be followed when weaving.

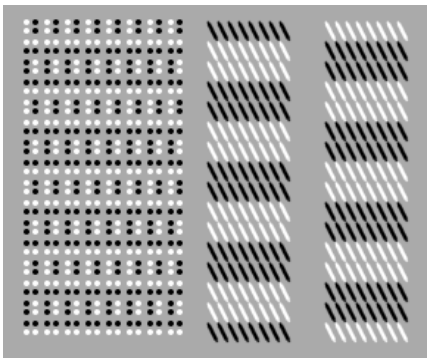

Figure 6: Simulated tablet weave for the single instruction `(weave-forward 16)`.

The language consists of simple instructions to represent the movement of the cards to create each shed. For example, Figure 6 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 Figure 7.

![Simulated tablet weave for the following:

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

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

](figures/07-diagonal.png)

One interesting limitation of tablet weaving is that it is not possible to weave 32 forward quarter rotates without completely twisting up the warp, so we need to go forward and backwards to make something physically weavable. However as Figure 8 demonstrates, if we do so then a 'zig zag' pattern results. Figure 9 shows that a different starting pattern that better matches the stitch direction.

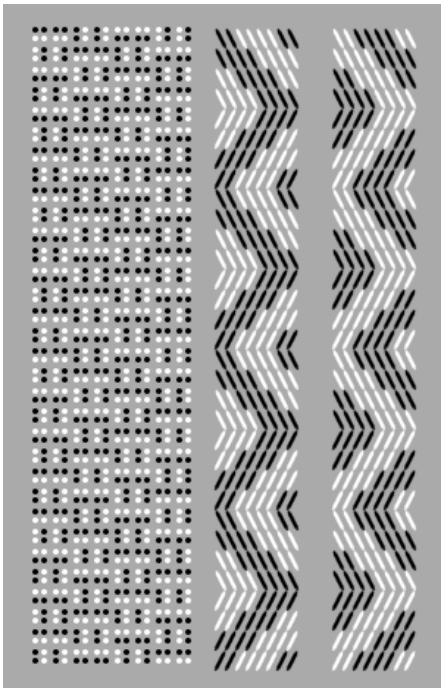

Figure 7: `(rotate-forward 0 1 2 3 4 5)` `(rotate-forward 0 1 2 3)` `(rotate-forward 0 1)` `(repeat 4 (weave-forward 4) (weave-back 4))`

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 by Figure 10. 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 in Figure 11.

![

Figure 8: `(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))`

```
(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))
```

](figures/10-mip.png)

xx

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.

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.

Flotsam Raspberry Pi Simulation

If we are to consider weaving as a digital technology, as it involves combinations of discrete elements or threads, then we can place it in the same category as technology that involves combinations of discreet voltages - the digital tools many of us use daily in our lives at present. When we do this we see many differences between the design of these tools, how we interact with them, and

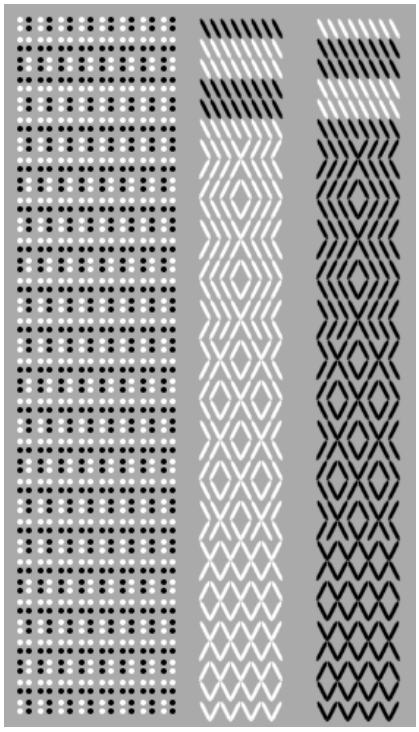

Figure 9: Following the same instructions as in Fig. 10, but where the first `(weave-forward 7)` is changed to `(weave-forward 6)` showing very different results.

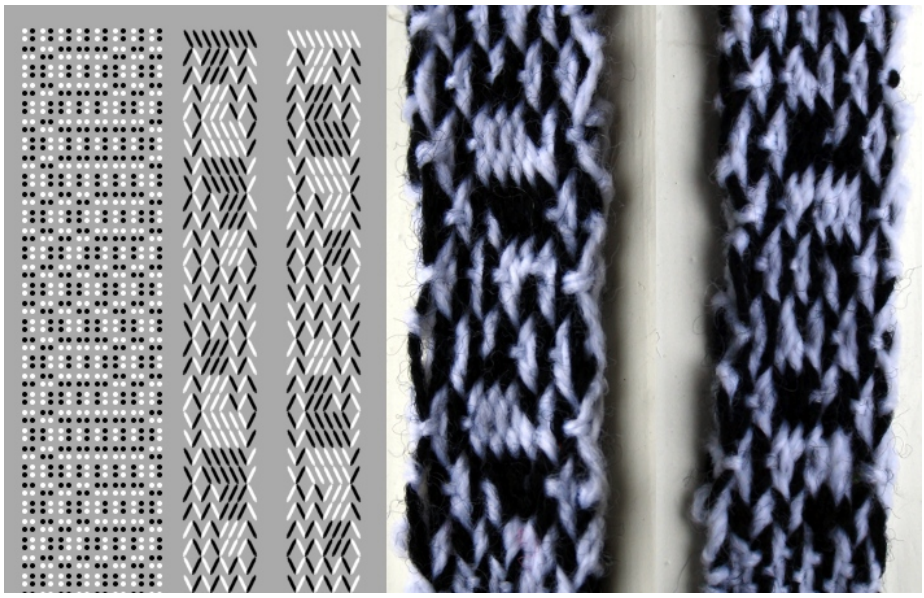

Figure 10: The following code, executed both by the simulation and by hand. (weave-forward 1) (twist 0 2 4 6) (repeat 4 (twist 3) (weave-forward 4) (twist 5) (weave-back 4)) “‘

their relationship to our bodies. One of the important strands of research on the WeavingCodes project turned out to be looking for how the design of weaving tools, having been honed over a long period of time - 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. It was constructed in order to experiment with new types of “tangible hardware” for teaching children programming without the need for a screen. 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 production rules for the warp/weft yarn sequences are constructed from the 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, 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 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 Minecraft

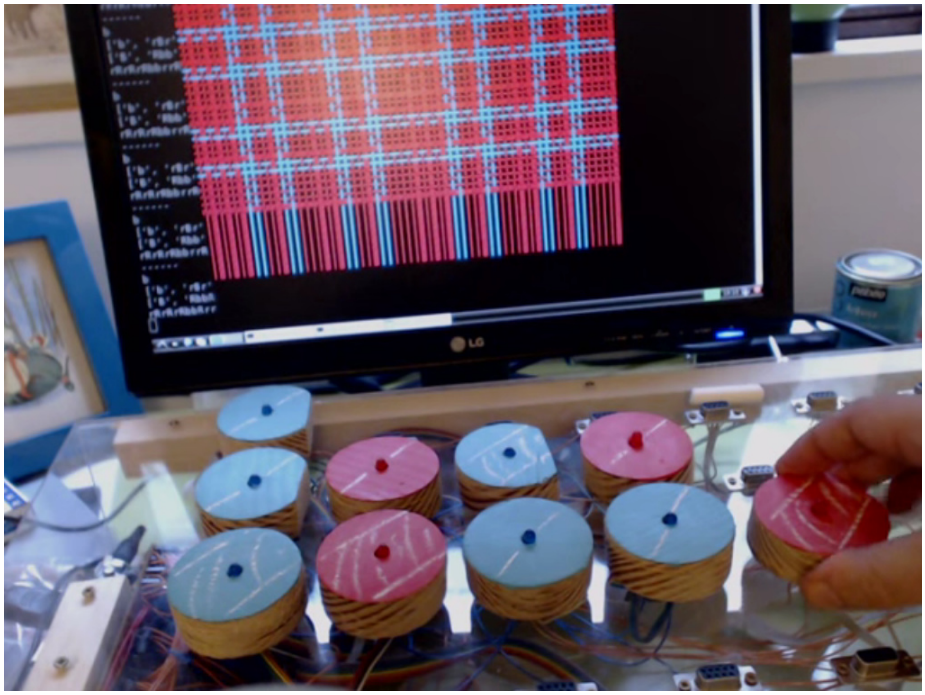

Figure 11:

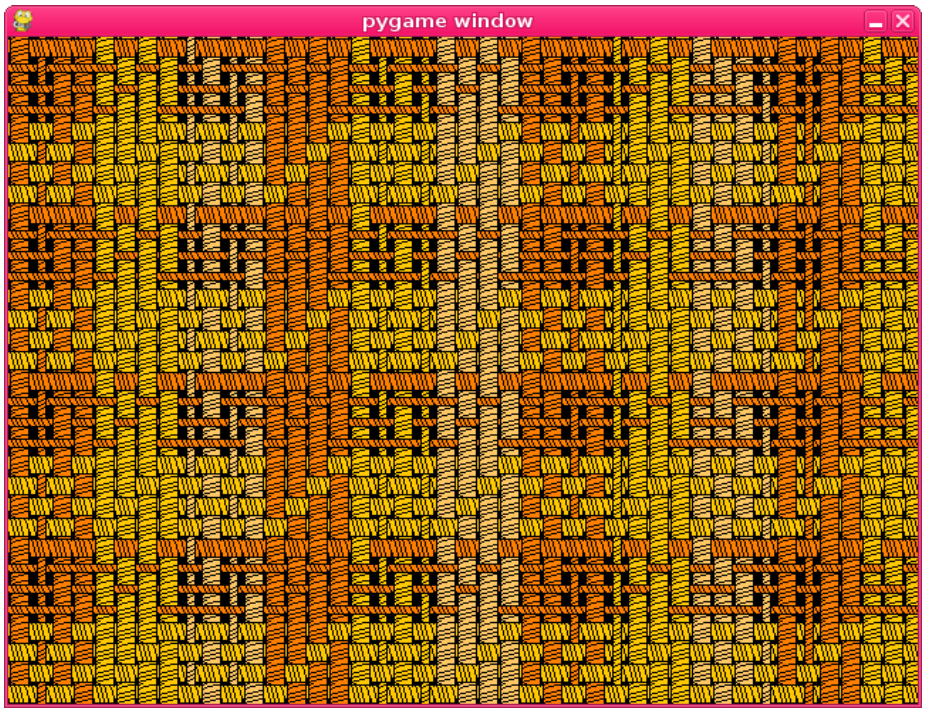

Figure 12:

3D game and a music synthesiser.

As before, an L system programming 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 design of the system itself needed further development, as the plugs were tricky to position correctly - particularly for small fingers.

However, during use we observed 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 interface encouraged collaborative learning beyond that possible with a traditional keyboard and screen interface designed for a single user.

Another aspect possible with this kind of interface is an increased role of touch - for example wrapping the tokens with the yarn that they represented (and the replication tokens in tinfoil) proved a very simple but effective 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 it much easier to explain.

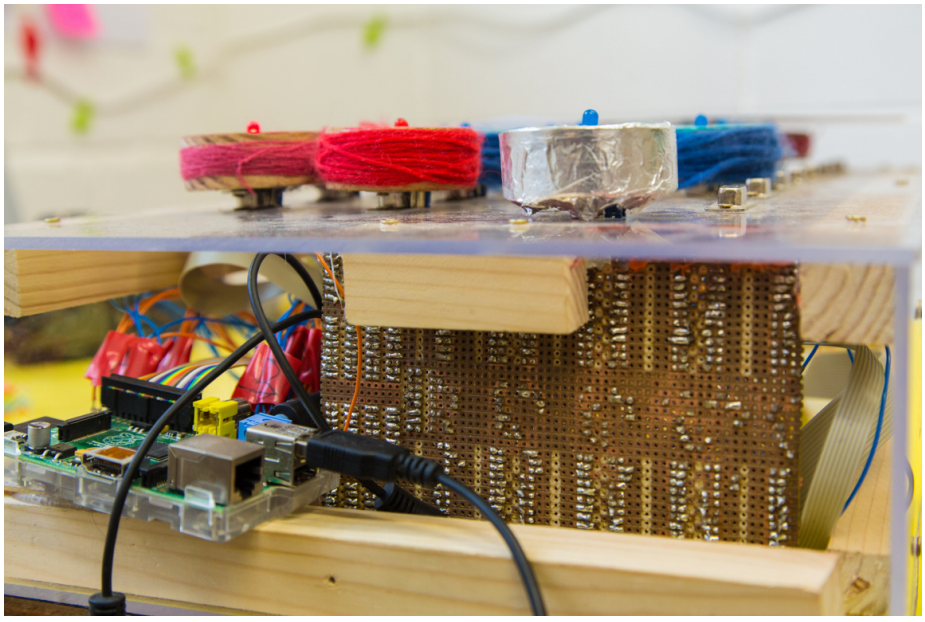

Figure 13:

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 in order to explore 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.

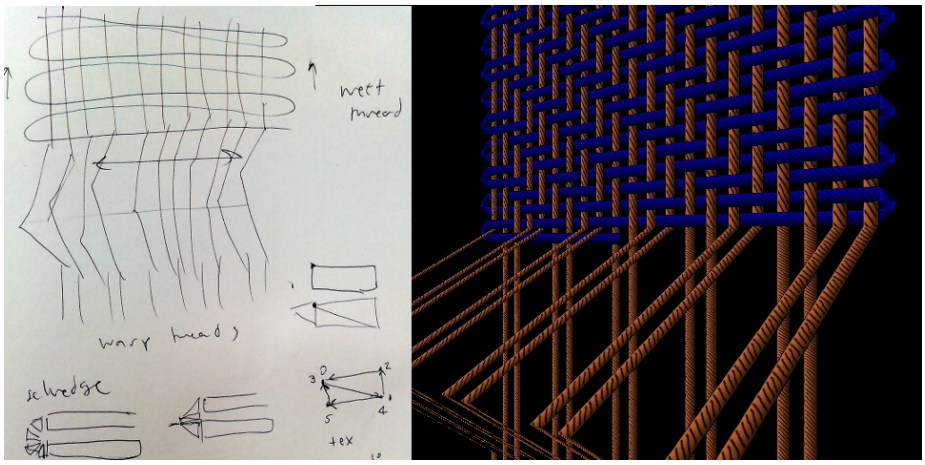

Figure 14:

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.

After our experience with Flotsam, we needed to explore tangible programming further. The pattern matrix was the next step, specialised for weaving and built by Makernow [http://www.makernow.co.uk/] [Oliver Hatfield, Andrew Smith, Justin Marshall] and FoAM Kernow as Open Hardware initially for use in 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. We then tested and developed it further in a museum exhibition 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, 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 availability 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

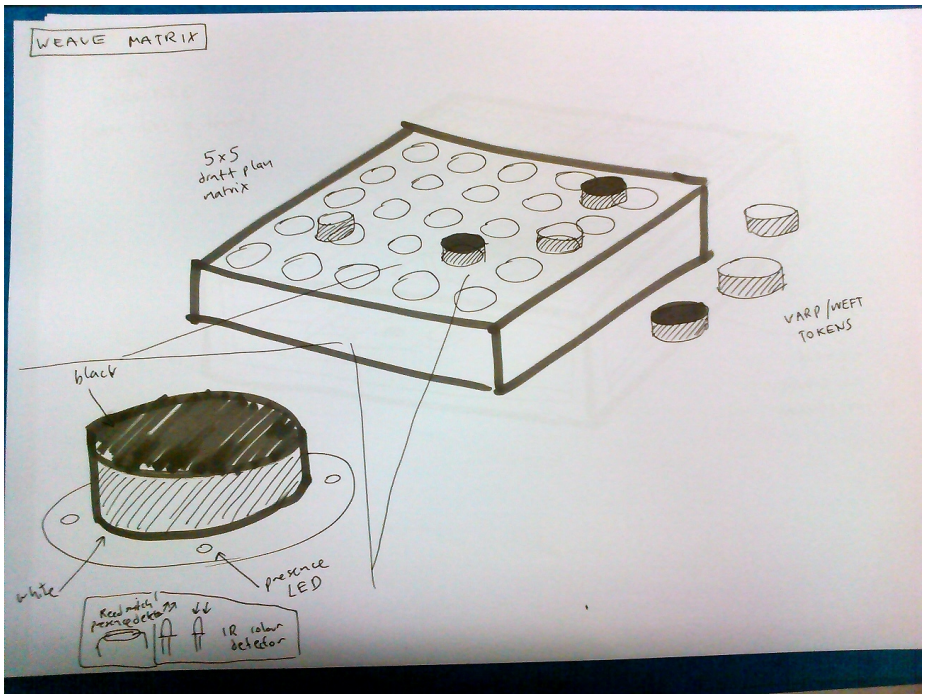
WEAVE MATRIX
5x5 draft plan matrix
WARP/WEFT TOKENS
black
white
presence LED
IR colour detector

Figure 15:

magnets in the centre and hall effect sensors in the base.

Hall effect sensors 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 (over or under), 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 block's magnets).

We used 4 hall effect sensors in a square which meant that we could detect rotation and flipping of the blocks. Incidentally, this has parallels with tablet weaving - in terms of the notation and the actions required to use the device. We can 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 and how they change their states with twisting and flipping:

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

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

A member of staff at Miners Court trying the first ever 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.

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

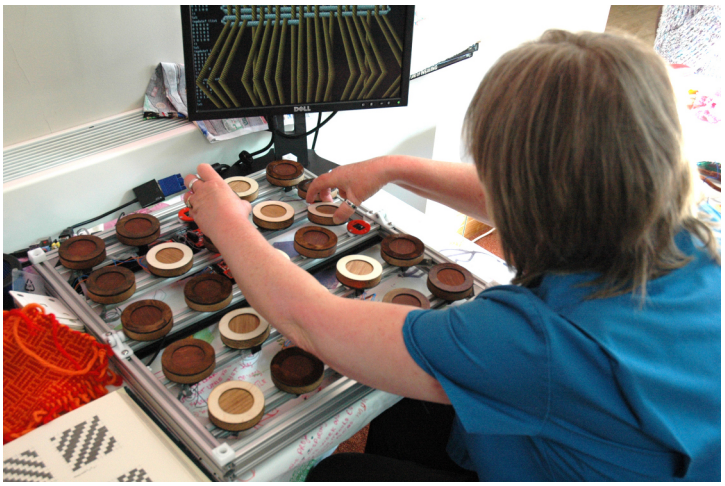

Figure 16:

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 3D 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 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 these complexities in this ancient weaving technique.

– Thread should be continuous?

type Thread a = Int -> a

data Unit = Warp | Weft

data Weave = Weave { }

– Weave as a raster data Weave = Weave {draft :: [[Bool]]}

– 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]

Figure 17:

– 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

– Weave as behaviour data Action = Pull Int | TurnIn | TurnOut | Over | Under type Weave = [Action] type Grid = [[String]] type Position = (Int,Int) data Move = TurnLeft | TurnRight | Straight data Direction = L | R | U | D type State = (Grid,Position,Move,Direction)

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 `Cow`s 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:

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

The number four would then be constructed with the following:

```
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.

```
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:

```
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
```

Dyadic browser weaving

compiled scheme code and rendering based on camouflage pattern research [?] [?]

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.

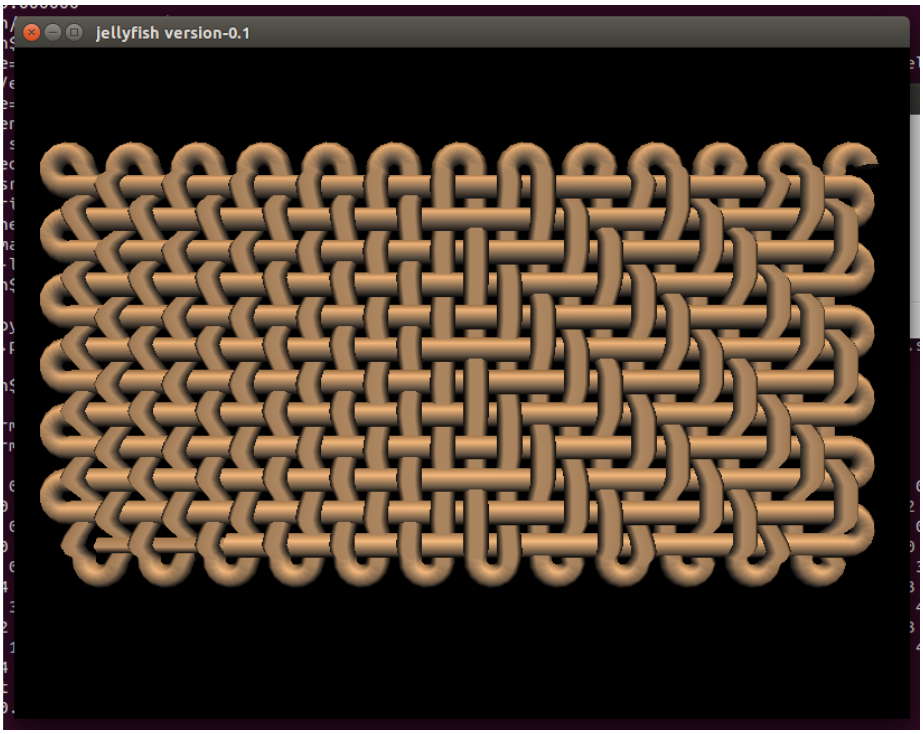

Figure 18:

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:

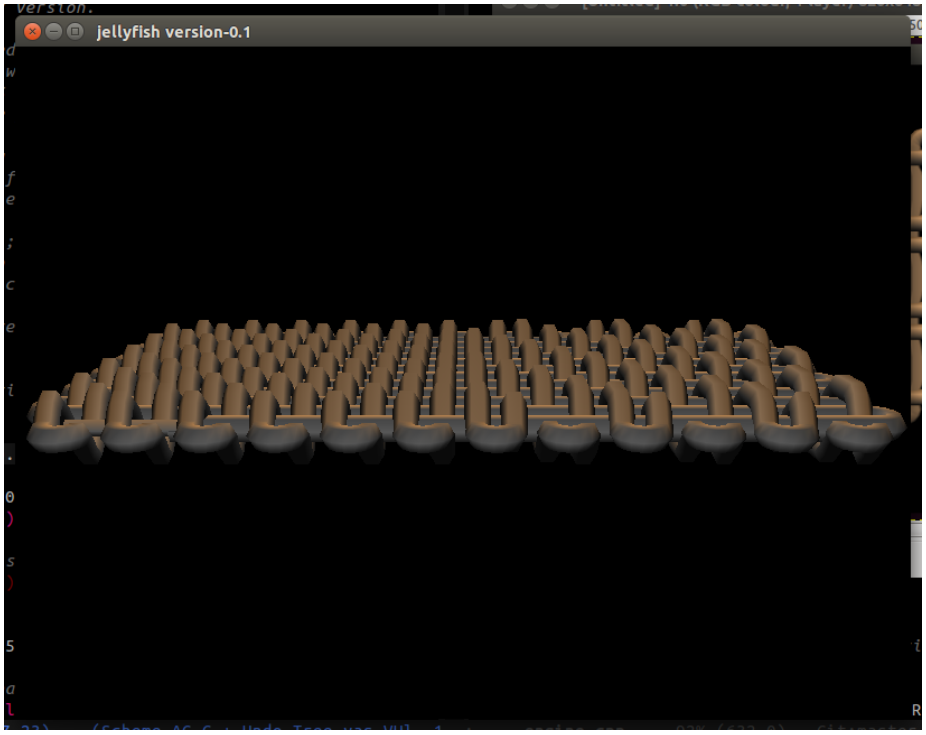

Figure 19:

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

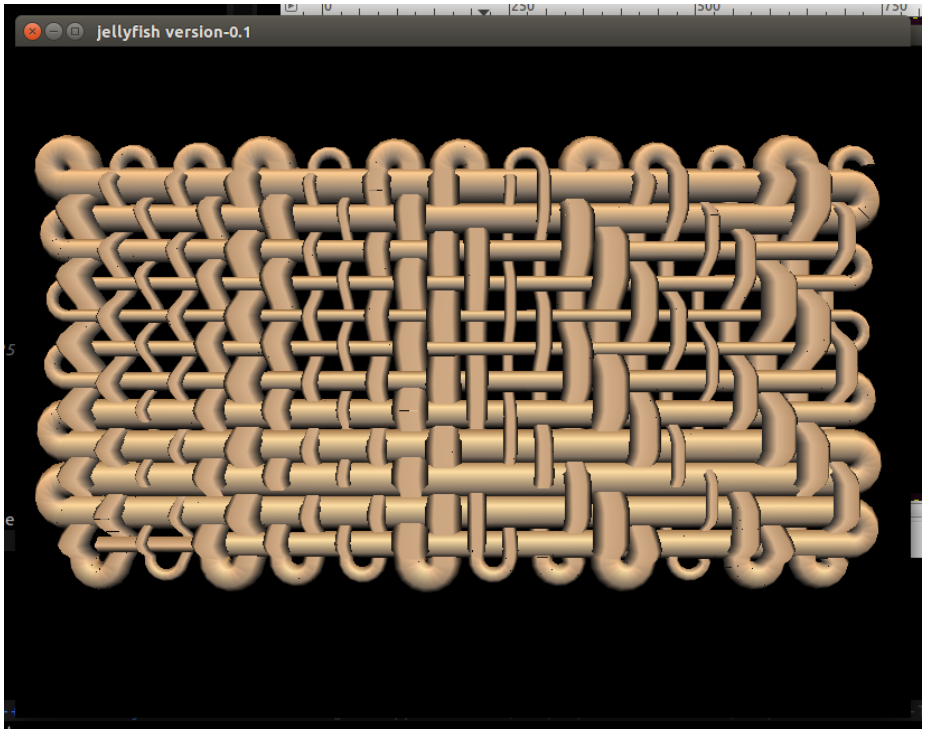

Figure 20: