A Minimal ZZStructure Navigator Using A ZigZag®-Style User Interface

John Ohno
Dr. Fischer
CS498

Table of Contents

Introduction

The minimal ZZstructure navigator described in this paper is a part of the Project Xanadu ® suite, and constitutes original work for software I developed at the request of Theodor Nelson, the originator of the project. In development since 1960, Project Xanadu ® aims to develop a suite of tools for creating, manipulating, and viewing media without the limitations enforced by skeuomorphism (particularly the limitations enforced by similarity to paper). ZigZag®, an implementation of which is provided here, is a part of this suite, and is a general purpose system for viewing and manipulating related pieces of data, consisting of both a front-end with standardized input operations and views, and a back-end with a standardized data structure.

Context

The ZZStructure is a data structure invented by Theodor Nelson[1]. It is general-purpose, and can represent a wide variety of patterns of association in a way readily visualized and navigated by a ZigZag® front-end. It is composed of a collection of objects called cells, each of which contain a value (a single piece of data, often a string or a number) and an associative array of pairs of pointers to other cells. Each pair of pointers may be considered a node in a doubly linked list. Each pair of connections in the array is called a dimension.[2]

ZZStructures can be modeled as a locally coherent but globally paradoxical topology (wherein systems more tangled than Klein bottles or Penrose triangles can be represented in more than three dimensions)[3]. They can also be modeled as a non-tabular relational system of data representation.[4]

Since a cell can have connections along a single dimension in two directions, the directions are

[1] See:

Nelson, Theodor. "A Cosmology for a Different Computer Universe: Data Model, Mechanisms, Virtual Machine and Visualization Infrastructure." Journal of Digital Information [Online], 5.1 (2004): n. pag. Web. 3 Jun. 2013

Nelson, Theodor Holm. "Interactive connection, viewing, and maneuvering system for complex data." US Patent 6262736. 17 July, 2001.

[2] For a less formal description, see:

Lukka, Tuomas. "A Gentle Introduction to Ted Nelson's ZigZag Structure." A Gentle Introduction to Ted Nelson's ZigZag Structure. The Free Software Foundation, 19 Dec. 2002. Web. 03 June 2013.

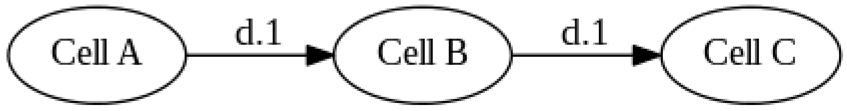
[3] See:

Michael J. McGuffin, m. c. schraefel. A Comparison of Hyperstructures: Zzstructures, mSpaces, and Polyarchies. Proceedings of 15th ACM Conference on Hypertext and Hypermedia (HT) 2004.

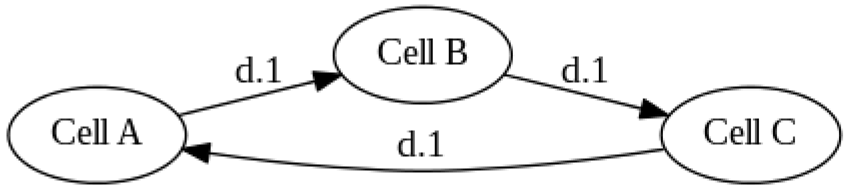
[4] See appendix E for more information on this application.

distinguished. One is called 'posward' and is considered to be forward-facing. The other is called 'negward'. A cell can be linked in one direction along some dimension, in both directions, or in neither.

In a ZZStructure, the sequence of cells formed by following a given dimension in both directions from a given cell is called a rank. In the 'multiple linked lists' model of the ZZStructure, a rank is a single linked list. Ranks can connect to themselves, forming what is called a ring rank. Ranks that are not ring ranks have a head and a tail; the head is the cell that has no negward connection along that dimension, and the tail is the cell that has no posward connection along that dimension.



A normal rank



A ring rank

ZigZag®[5] is a ZZStructure packaged with a particular set of operations and a particular interaction model. Specifically, ZigZag® implementations have pluggable visualizations of the ZZStructure (called views), a default set of operations with associated keyboard assignments called KBLANG, and the concept of one or more moving cursors that focus the view on a particular cell.

In the context of ZigZag®, the set of cursors is called a cursorplex, and the cell pointed to by a cursor is called the accursed cell. The cursorplex is important for views, and most KBLANG operations manipulate the cursors or the accursed cells.

Several common views exist. All of them are dual-pane, meaning that there are two windows, each showing the same ZZStructure, possibly with a different accursed cell and different visible dimensions. Dual-pane views are convenient for implementing core KBLANG, because it has two sets of reserved navigation keys and a single set of mutator keys.

In grid view, cells are laid out in a grid, showing links along two dimensions at right angles to each other, with the accursed cell in the centre and all cells the same size. Stretch-vanishing view is similar, but overlapping cells are avoided by spreading rows or columns of cells apart, even when they are not connected, and cells are resized to fit their content.

[5] ZigZag® is a registered trademark of Theodor Nelson

ZigZag® has a handful of 'special' dimensions, for special kinds of operations. An important one is the 'clone' dimension: all cells linked together along the clone dimension share the same value but have different connections.

There have been many implementations of ZigZag®. Aside from the implementation that forms the subject of this paper, I will cover only RZZ in depth. For a brief description of the others, and for more information about RZZ, see appendix B.

A brief history of the ZigZag® project

The ZigZag® design was invented in 1983. The first full implementation was AZZ, by Andrew Pam, in 1997. The second was GZZ, in 2000, by Tuomas Lukka and his associates. The third was LZZ by Les Carr. In 2003, EZZ was implemented by Mikhail Seilverstov, and ZZZ was implemented by Jeremy Smith. In 2006, RZZ was implemented by Robert Smith.

RZZ was written entirely in C++, with a python binding. It was used as the basis for a large amount of code written in python. The python binding was not object oriented, and relied heavily upon the use of tuples acting as unique identifiers for cells. It had several interesting optimizations, most notably the Synapse.

The Synapse was an object used in an optimization existing only in RZZ, for the purpose of reducing the time complexity of finding the head and tail of a rank from linear to constant time. This optimization is useful when making heavy use of clone cells, because the value of a clone cell is stored in the head of the clone rank containing that cell. Each Synapse object stored the head and tail of a rank, and every cell contained a pointer to a Synapse object for every dimension. While this reduced all head lookups to constant time, it caused all rank joins and rank splits to exhibit linear time behavior with respect to the number of cells in the rank.

Jon Kopetz and I were brought onto the project in August of 2011 to document and debug RZZ. The implementation presented is similar to RZZ because of backwards-compatibility concerns. I originally wrote the front-end presented in this document in January of 2012 as a tool for debugging RZZ. Jon Kopetz's primary contribution to the attached code is the implementation of the DimFactory class and the method used for tuple-masquerading, though his contribution is much greater in the larger project of which this is a part. My role in this development is voluntary and unpaid.

Goals

My project was to implement a general purpose ZZStructure library and a user interface for viewing, navigating, and editing a ZZStructure using it. My view was to be a two-dimension dual-pane grid view. I was to implement all the core KBLANG operations and hook them into their associated key bindings. In this way, my application could be used by users of other ZigZag® implementations, and my back-end could be used for the purpose of general-purpose storage.

I was to clearly distinguish the accursed cells from other cells, and indicate the accursed cell from one pane in the other pane for operations that involved both panes, and provide simple and comprehensible help information for new users[6].

The goal of Project Xanadu ® as a whole is to produce the entire suite of software of which this is a part, including software for lossless drafting of documents with unbreakable links, a micropayment framework, and a set of flexible visualization systems. This suite is planned to become commercially available.

[6] See Fig 3

Design decisions

I decided to implement both my front-end and back-end in python[7]. While python is slower than compiled languages (and slower than many other interpreted languages), I determined that the slowness would not be extreme enough to negatively impact usability. Python has a large standard library, easy to use built-in implementations of associative arrays, and a quick debugging turnaround. It can be prototyped from an interactive interpreter, and has a large amount of good documentation available.

Other implementations used perl (AZZ), Java (GZZ), C++ (ZZZ and RZZ), or javascript (LZZ).

I did not use perl because it is difficult to keep high standards of code readability in that language, and it offered no benefits over python. Perl also has complex rules for associative array indexing -- which my code does very often -- and certain elements of its syntax for this make it inconvenient to store heterogenous data types in associative arrays. Additionally, perl has many competing object orientation systems, none of them fully standard. Also, unlike python, perl has exactly one full implementation, and compatibility is rarely maintained between releases.

I did not use Java in part because Java is rapidly losing support. Other than cellular telephones and cable set-top boxes, new computers are not coming with Java installed. Java recently changed owners, and Oracle may sell it again or make major changes, or simply drop the product line. While several Java implementations exist, they are all forked from the same codebase (with the exception of GCJ, which doesn't have support for features released after version 1.5 and doesn't contain some parts of the standard library). Furthermore, while a ZZ backend in java is trivial, a ZZ front-end in java would be awkward to implement, and java is inconvenient for iterative development. GZZ was highly successful, but it was worked on by a large team over a long period.

I did not use C++ because writing cross-platform graphics code in C++ is difficult. While writing the back-end in C++ is trivial, the front-end is not. Furthermore, work with RZZ demonstrated that having a C++ backend with a python frontend is counterproductive: the backend must be re-compiled for every platform, and the overhead of passing over the python bindings (and performing type checking and type conversion there) is not significantly lower than the overhead of simply having a backend written in python.

The history of LZZ discouraged me from implementing this project in javascript. LZZ was, at the time of the initial release, not functional in all browsers. It does not work in any current browser. While javascript can be run standalone, this is unusual -- the only javascript environment most

[7] Python™ is a trademark of the Python Software Foundation. I used the language version 2.7 as defined in http://docs.python.org/2.7/reference/index.html

people have is built into their browser, and it rarely conforms to any standard.

I decided to fully separate my front-end from my back-end, and make my back-end completely standalone. The advantage of such a modular approach is that the ZZStructure implementation can be reused in other projects where ZZStructures are needed. I maintained compatibility with the python API provided by RZZ.

Unfortunately, RZZ did not use objects, and instead passed tuples back and forth between python and C++, assuming that tuple size uniquely specified object type[8]. While this makes some sense in a C++ implementation (because a friendlier API would be written in slower python code and wrap the C++ bindings), in a pure python implementation it simply means that object orientation cannot be used. So, instead, I made regular python objects that behave like tuples when indexed. My versions of the API functions will take tuples or the corresponding objects, and will always return objects; tuples will be converted to objects before they are first used, and then object member functions are used. In this way, I kept compatibility with an ill-designed API and the large amount of code that uses it, without having to subject myself or my users to the same design failures.

Of the existing implementations, only RZZ had a strict separation between front-end and back-end. Tight coupling means that there can be a one-to-one correspondence between KBLANG keys and back-end functions, but this is at the cost of duplicating code. Having the back-end completely separate means that more novel views can be constructed, and also that applications of ZZStructures can be employed that take advantage of automation (such as the database use described in appendix E).

I decided to use Tkinter as my GUI library. Tkinter is a python wrapper around the popular TCL binding to the GUI/widget library TK. I chose TK because I was familiar with it and because it is the only GUI library that comes with nearly every distribution of python. However, I used few of the features provided by TK: I primarily relied upon the canvas and binding system.

While Tkinter is implemented in a fairly usable way, the idiomatic way of using TK from TCL is very different from idiomatic python, and idiomatic Tkinter lies somewhere in the middle. Tkinter attempts to wrap object orientation around a TCL idiom based on hard-coded magic strings, string 'tags' used as non-unique identifiers, and non-function 'commands' that create new 'commands' that act like objects but take tag arguments. Tkinter fully fails to integrate the canvas object into its object oriented model, because the canvas contains a large number of similar but not exactly equivalent versions of other widgets, controlled using tags sent to the canvas object.

[8] Cells had unique descriptors consisting of two parts: a 3-tuple of strings called the slice, and a positive integer called the CID. I overloaded most API functions that took a slice and a CID to also take an FQCID, which is a 4-tuple consisting of the CID at the end of the slice; cell objects masquerade as FQCIDs, so this allowed them to be passed in as single objects instead of being sliced by a front-end developer.
The larger ecosystem of python code of which RZZ was a part contained several other similar structures, all represented with tuples, whose sizes were not unique across modules.

The idiosyncrasies of Tkinter cost me many hours of debugging time, but provided me with the capability to write truly cross-platform GUI code with minimal dependencies.

Other common widget libraries for use with python include Qt, SDL, and Wx.

Qt is very popular and cross-platform. I did not use it, because Qt has its own complex ecosystem (including its own set of types). Once someone begins a Qt project, it is difficult to interoperate with code written without Qt in mind. On the other hand, Qt has a large number of useful features that Tk lacks: support for media like videos, a system by which widgets can be easily arbitrarily extended. Qt widgets are consistently object oriented in their use, and the python binding to Qt is fully idiomatic python. However, even programs in interpreted languages like python need special Qt-specific build tools in order to work properly when using Qt. There is a fantastic ZigZag-like system written by Jonathan Kopetz in python using Qt, called Dimscape[9]; it uses the Qt features missing from Tk to great effect, having not only video support but a novel view with curved, fluidly animated dimension lines.

SDL is also popular, and it is in a sense more cross-platform than Qt (in that it also runs on video game consoles, cellular telephones, and obscure/obsolete operating systems). However, SDL is not properly a widget library; it is merely a graphics library. While it is possible to use SDL to implement a view (both RZZ and ZZZ used it), the developer must roll his own implementations for things like bounding boxes, z-levels, draw lists, and animation tweening. Furthermore, while SDL is fully cross-platform and python is fully cross-platform, SDL bindings to python are not necessarily fully cross-platform.

Wx is popular, and supports both Windows and X. However, it suffers from many of the same problems as mentioned in connection with Qt, without being quite as popular or featureful.

On my back-end, I decided to use several lazy optimizations.

The operation for retrieving the head of a rank can be linear time; RZZ avoided this by keeping around another object called a Synapse or a RankNode[10] (discussed in the history section, and in appendix B under RZZ). I consider a linear time penalty on links unreasonable.

Instead, I have a lazy head-finding optimization: when the head is requested, a cached head is checked for; if the cached head has a negward link on the given dimension, the rank is traversed negward until either no negward link is found (in which case, we have found the correct head, and we traverse the rank posward updating the cells) or we reach the original calling cell (in which case we have a ring rank, which has no head). A cached head cell with no negward link is up to date, and is returned in constant time. The same method is used for determining the tail of a rank.[11]

[9] See appendix B for more information on DimScape

[10] This was a feature of RZZ.

[11] This method is subject to a potential exploit, wherein head is called immediately before a rank is broken

In my front end, I decided to go with a view model based on method overriding. My default (grid) view would be a single class, with a single core (recursive) drawing function. Other views would minimally override this recursive drawing function and replace it with fresh drawing code. This differs from GZZ, where all views inherited from a single abstract view, despite the fact that most views differed only in how the placement of cells was calculated.

I decided to support both clipping and a time-to-live value into my recursive drawing function, and have it draw exactly one cell at a time, primed with a dimension and a direction. Were I to end recursion on clipping only, there is the possibility that a pathologically designed structure could overflow the stack without ever hitting the edge of the window; the time-to-live value provides a hard limit to the stack size. Limiting each call to drawing exactly one cell provided the simplicity necessary to make an already extremely complex process of adjusting positions debuggable.

Difficulties

Early in development, a design was in play in which the back-end was connected to the front-end via a socket. The protocol was similar to KBLANG, only stateless: all sets of cells had to be specified. In this way, the server could effectively handle many users simultaneously editing the same ZZStructure in parallel. However, this part of the design was dropped because during a test wherein the front end and back end were connected over the internet between California and Connecticut, round-trip lags slowed redraw time and responsiveness to the point where every operation had a quarter second delay. Caching was considered, and it was decided that a caching system that operated appropriately would be indistinguishable from a redundant copy of the backend with added synchronization code.

RZZ’s API relied heavily on autovivification. While autovivification of cells was easy to implement, the logic to create brand new cells with unknown identifiers interfered with autovivification code a number of times in implementations of functions that should never create new cells. This problem was solved by separating the functionality into several different functions, one of which autovivified cells, another of which created new cells, and a third of which merely turned tuples into cell objects and failed if the cell did not exist, and using these more specialized functions in most of the internal code.

---

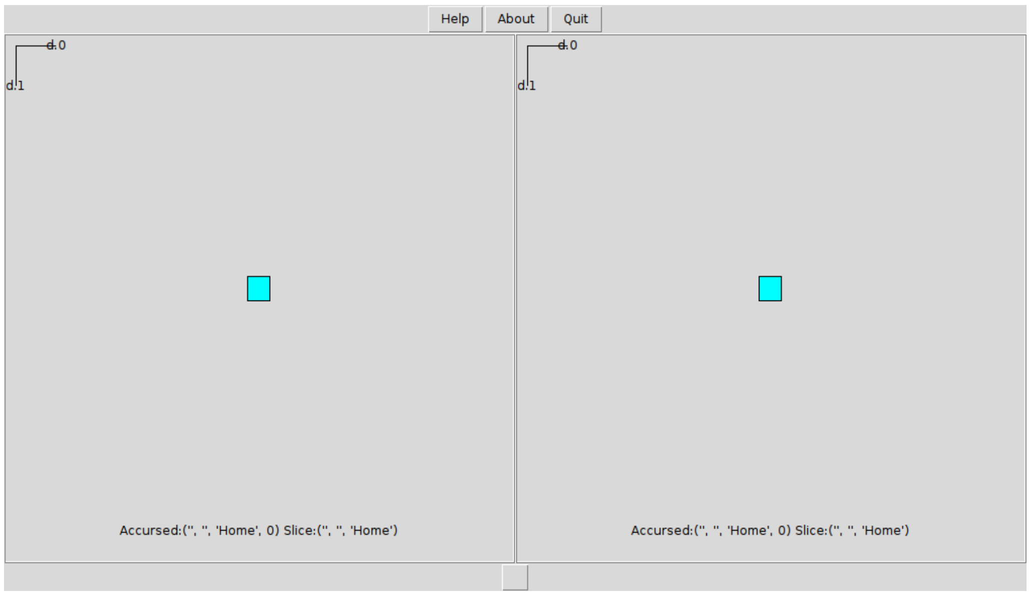
and reconnected, and a subsequent call to head gets the wrong value. A potential solution to this is to perform another pass -- instead of updating the cached head cell on the way back down, wait until you reach the tail cell, and then update on the way back up. If you do not pass the cell where head was called, there’s a broken rank. This remains linear time and behaves more consistently, but triples the coefficient at the least. The potential bug was considered too unusual to justify incurring this time penalty on every head operation with a stale cached head. Since a fix is known, it can be implemented later.

Results

I have invested roughly 1600 hours into Project Xanadu as a whole, with the attached ZigZag® implementation alone consuming a tenth of the total (160 hours). It may take another forty or fifty hours of work to bring this subproject to total completion.

Were I to start over with the knowledge I have now, I would spend more time writing generalized wrappers around Tkinter to protect myself from some of the inconsistencies of that code; furthermore, knowing now that most RZZ-dependent code is going to be phased out and re-written, I would drop tuple support from the beginning and focus entirely around cell objects, avoiding the existing API and its general structure and assumptions. I would not support autovivification. Since conflicts between Tkinter and python idioms and problems relating to tuples and autovivification together account for the majority of debugging time, this would shorten the project's difficulties greatly.

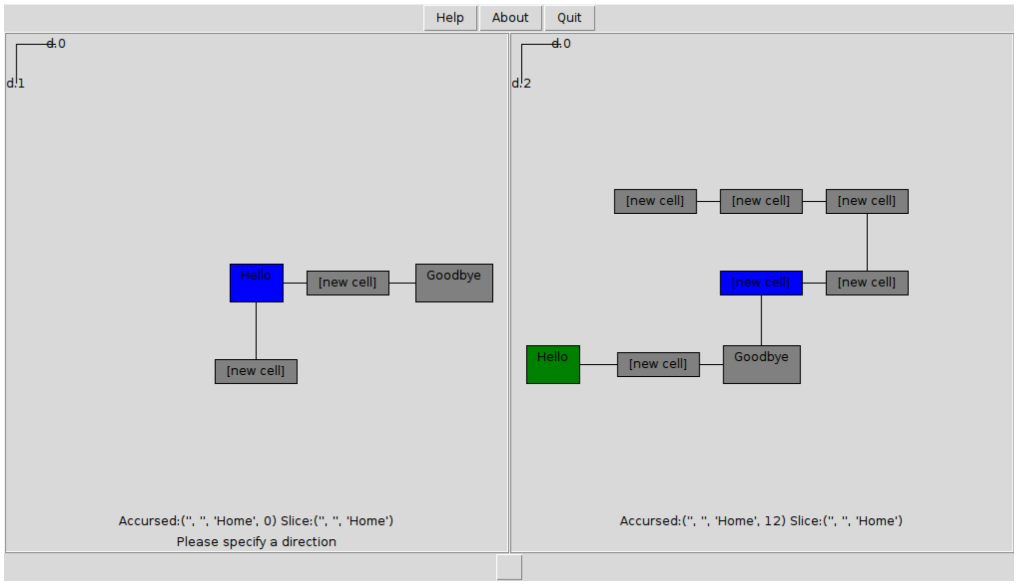
The attached implementation provides full navigation and editing support with KBLANG.



```
Accursed:('', '', 'Home', 0) Slice:('', '', 'Home')
```

```
Accursed:('', '', 'Home', 0) Slice:('', '', 'Home')
```

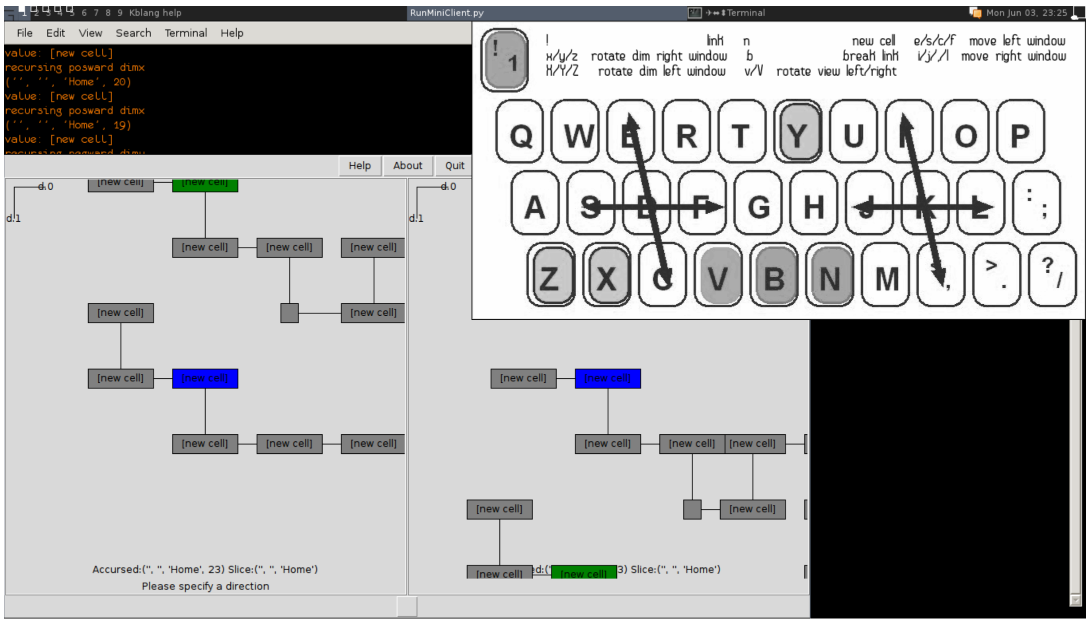
Fig 1 - A blank ZZstructure



```
Accursed:('', '', 'Home', 0) Slice:('', '', 'Home')
Please specify a direction
```

```
Accursed:('', '', 'Home', 12) Slice:('', '', 'Home')
```

Fig 2 - A ZZstructure with more content. Note the differently colored cursors. Some content is not visible in the left pane because of clipping rules in the recursive drawing routine.



```
value: [new cell]
recursing posward dimx
('', '', 'Home', 20)
value: [new cell]
recursing posward dimx
('', '', 'Home', 19)
value: [new cell]
```

```
!                               link    n                     new cell   e/s/c/f   move left window
x/y/z   rotate dim right window  b                           break link  i/j/,/l   move right window
X/Y/Z    rotate dim left window  v/V   rotate view left/right
```

```
Accursed:('', '', 'Home', 23) Slice:('', '', 'Home')
Please specify a direction
```

Fig 3 - A fairly complicated structure, with the help window open over it.

Future work

A projected 40-50 additional hours of development will be invested in the front-end, and the back-end will continue to be improved indefinitely in the context of related projects. This is an ongoing project.

My ZZStructure implementation is currently at the core of another Project Xanadu®[12] related project, as the in-memory representation of text layout in an implementation of the system described in (NELSON-4)[13] superseding the implementation that used RZZ, and other front-ends are being implemented for it.

There are plans to use the system as a general purpose database. There are also plans to implement the system described in (NELSON-3)[14].

An on-disk storage format for ZZStructures has been proposed, based off a subset of YAML[15]. One benefit of this implementation over earlier implementations is the cleanly documented API. While it shares RZZ's API, the documentation for RZZ's API did not exist before I reverse-engineered it -- so, in a sense, this is the first implementation whose back-end can be used by a third party without intense consultation.

Another benefit of this implementation is that it takes advantage of a popular cross-platform language in a way that will continue to function for the forseeable future, even in later versions. AZZ used perl, an interpreted language -- however, the perl code needed to be changed in order to support windows. RZZ was theoretically cross-platform (C++ can be bound to python on a wide variety of platforms), but the python bindings were specific to a particular version of python, and newer versions of python will not bind to RZZ. My implementation, on the other hand, happily runs on python 2.6, 2.7, and 3.

---

[12] "Xanadu®" is a registered trademark of Project Xanadu.

[13] Nelson, Theodor Holm. "System for exploring connections between data pages." Patent US20090222717 A1. 3 September, 2009.

[14] Nelson, Theodor Holm. "System for combining datasets and information structures by intercalation." Patent US20050192981 A1. 1 September, 2005.

[15] The format in question can be represented as BNF as follows:

```
<CID> := [0-9]+
<DIM> := [a-z][a-z]*”.”[a-z0-9][a-z0-9]*
<span> := ([0-9][0-9]*”-”[0-9][0-9]*)(“,”[0-9][0-9]*”-”[0-9][0-9]*)*
<cell> := <CID>”:”<span><span>*(“\t”<DIM>”:”<CID>)*
```

Bibliography

Nelson, Theodor. "A Cosmology for a Different Computer Universe: Data Model, Mechanisms, Virtual Machine and Visualization Infrastructure." Journal of Digital Information [Online], 5.1 (2004): n. pag. Web. 3 Jun. 2013

Nelson, Theodor Holm. "Interactive connection, viewing, and maneuvering system for complex data." US Patent 6262736. 17 July, 2001.

Nelson, Theodor Holm. "System for combining datasets and information structures by intercalation." Patent US20050192981 A1. 1 September, 2005.

Nelson, Theodor Holm. "System for exploring connections between data pages." Patent US20090222717 A1. 3 September, 2009.

Michael J. McGuffin, m. c. schraefel. A Comparison of Hyperstructures: Zzstructures, mSpaces, and Polyarchies. Proceedings of 15th ACM Conference on Hypertext and Hypermedia (HT) 2004.

Lukka, Tuomas. "A Gentle Introduction to Ted Nelson's ZigZag Structure." A Gentle Introduction to Ted Nelson's ZigZag Structure. The Free Software Foundation, 19 Dec. 2002. Web. 03 June 2013.

Appendix A: Class list

Front-end module

```
def dprint(foo, d=1):
class ZZView:
    Other view classes should inherit from this.
    def __init__(self, master=None, width=500, height=500, brother=None):
    def goHome(self):
    def dirty(self):
    def clearPane(self, args=None):
    def drawRose(self, args=None):
    def reDraw(self, args=None):
    def rotDimX(self, args):
    def rotDimY(self, args):
    def draw_r(self, distance=25, cell=None, pos=-1, dim="", drawx=None, drawy=None, marked=[], cellloc={}):
    def chug(self, dim, pos):
    def chugSouth(self, args):
    def chugNorth(self, args):
    def chugEast(self, args):
    def chugWest(self, args):
    def editCell(self, args=None):
            def tempCallback(args=None):
    def insertCell(self, args):
    def deleteCell(self, args):
def displayHelp(args=None):
def displayAbout(args=None):
def loadUI():
    def paneCreateHelper(args=None):
    def paneChugLeft(args=None):
    def linkHelper(args=None):
    def unlinkHelper(args=None):
    def goHomeHelper(args=None):
```

Back-end module

```
class DimFactory(object):
     def __init__(self, prefix="d."):
    def __getattr__(self, name):
    def __call__(self, name):
    def __getitem__(self, name):
    def __len__(self):
    def dims(self):
class Dim(object):
    def __init__(self, dim):
    def normalize(self, val):
    def slice(self):
    def cell(self):
```

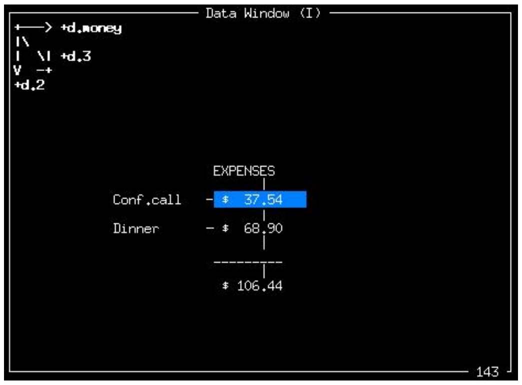
```
    def reverse(self):
    def __contains__(self, needle):
    def __getitem__(self, idx):
    def __getslice__(self, i, j):
    def __iter__(self):
    def __len__(self):
    def __neg__(self):
    def __pos__(self):
    def __invert__(self):
    def __str__(self):
    def __repr__(self):
    def __hash__(self):
    def __eq__(self, oth):
    def __neq__(self, oth):
    def __gt__(self, oth):
    def __ge__(self, oth):
    def __lt__(self, oth):
    def __le__(self, oth):
    def __cmp__(self, oth):
class CellFactory(object):
    def __init__(self):
    def slice(self, aSlice):
    def __call__(self, name):
    def __getslice__(self, maybeSlice, maybeCell):
    def __setslice__(self, maybeSlice, maybeCell):
    def __delslice__(self, maybeSlice, maybeCell):
    def __delitem__(self, name):
    def __getitem__(self, name):
    def __setitem__(self, idx, val):
    def __len__(self):
        def __init__(self, dim):
        def __contains__(self, needle):
        def __getitem__(self, idx):
        def __getslice__(self, i, j):
        def __iter__(self):
        def __len__(self):
        def __str__(self):
        def __repr__(self):
        def __eq__(self, oth):
        def __neq__(self, oth):
class Slice:
    def __init__(self, f='', server='', account=''):
    def __contains__(self, a):
    def __getitem__(self, a):
    def __iter__(self, *a):
    def __len__(self, *a):
    def __setitem__(self, a, b):
    def keys(self, *a):
    def items(self, *a):
    def iterkeys(self, *a):
```

```
    def iteritems(self, *a):
    def itervalues(self, *a):
    def values(self, *a):
class Cell:
    def __init__(self):
    def name(self):
    def __repr__(self):
    def __len__(self):
    def __getitem__(self, item):
    def getName(self):
    def fqcid(self):
    def next(self, dim):
    def setnext(self, dim, cid):
    def prev(self, dim):
    def gethead(self, dim):
    def clonehead(self):
    def getValue(self):
    def setValue(self, v):
    def insert(self, dim, value):
    def remove(self, s, cid):
def slices():
def cells(sl):
def slice(tup):
def cell(tup, idx=None):
def createslice(tup):
def createcell(sl=(", ", "), content="", cid=-1):
def destroyslice(sl):
def destroycell(tup):
def value(cid):
def setvalue(cid, val):
def executecell():
def createclone(cid):
def clonehead(tup):
def clones(tup):
def setclonehead():
def next(cid, dim):
def prev(cid, dim):
def last(cid, dim):
def following(cid, dim):
def brek():
def edit(cid, dim, *val):
def insert(cid, dim, *val):
def append(cid, dim, cid2):
def rank(cid, dim):
def head(cid, dim):
def remove():
def dimkeys():
def dimkey(dim):
def reverse():
def createdimkey(dim):
```

```
def undo():
def redo():
def saveslice():
def loadslice():
def saveslicelog():
```

Appendix B: Other implementations

AZZ
AZZ was implemented in Perl by Andrew Pam. Its user interface is purely textual, using curses. It uses a grid view, and does not de-duplicate cells in ring ranks. It initially had support only for Linux, then later was distributed as a bootable floppy disk image containing a minimal Linux distribution and a copy of perl (http://xanadu.com.au/zigzag/zzdemo.zip). Eventually, it was ported to a variant of perl on Windows 95.
Original AZZ source: http://xanadu.com.au/zigzag/zigzag-0.70.tar.gz



```
Data Window (I)
+——> +d.money
|\
|  \| +d.3
V  -+
+d.2


                    EXPENSES
                        |
          Conf.call  - $  37.54
                        |
          Dinner     - $  68.90
                        |
                    ---------
                        |
                     $ 106.44

                                      143
```

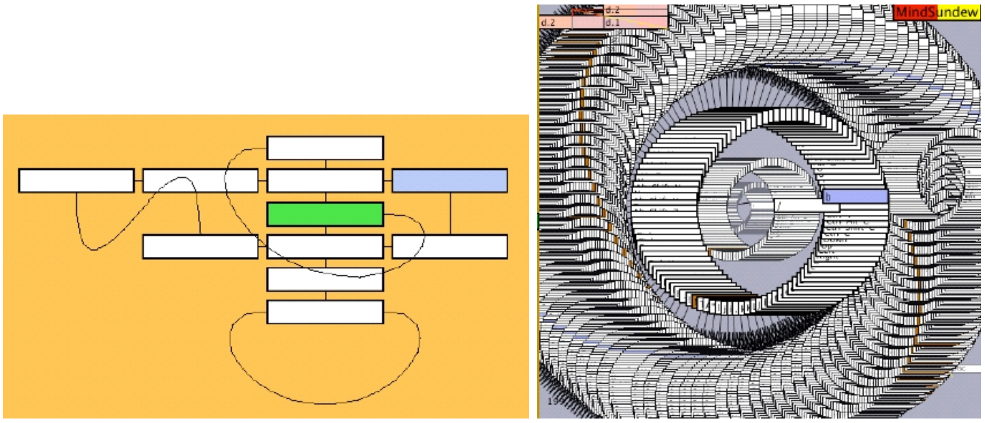
A typical screen in AZZ, from Ted Nelson's 1998 talk "What's on My Mind"

GZZ
GZZ was implemented by Tuomas Lukka and some of his colleagues. It was originally called GZigZag. The project stopped due to intellectual property disagreements, and some of the developers went onto a project called FenFire.

GZZ is still the most popular implementation, and a pre-packaged demo exists containing GZZ and a handful of sample data sets: http://xanadu.com/zigzag/zzStarterKit.zip

Adam Moore has produced an interesting video, demonstrating the capabilities of GZZ: http://xanadu.com/zigzag/ZZdnld/zzChemDemo-Moore.mov

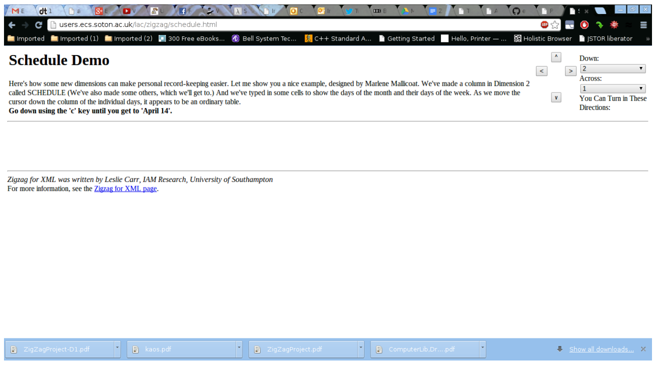
One particularly notable thing about GZZ is that it supported a wide array of novelty views.



Left - a typical ZZStructure in the grid view in GZZ. Right - the most popular novelty view, “MindSunDew”

LZZ

LZZ can still be accessed at this location: http://users.ecs.soton.ac.uk/lac/zigzag/



Schedule Demo

Here's how some new dimensions can make personal record-keeping easier. Let me show you a nice example, designed by Marlene Mallicoat. We've made a column in Dimension 2 called SCHEDULE (We've also made some others, which we'll get to.) And we've typed in some cells to show the days of the month and their days of the week. As we move the cursor down the column of the individual days, it appears to be an ordinary table.
Go down using the 'c' key until you get to 'April 14'.

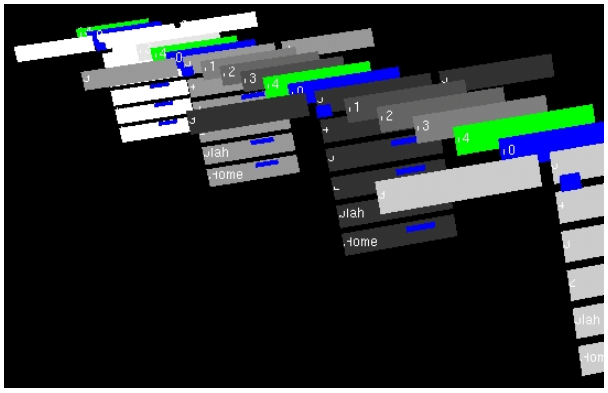
Down: 2
Across: 1
You Can Turn in These Directions:

Zigzag for XML was written by Leslie Carr, IAM Research, University of Southampton
For more information, see the Zigzag for XML page.

LZZ, ‘running’ on a modern browser

ZZZ

ZZZ, written by Jeremy Smith, was the first implementation to project a ZZStructure in three dimensions rather than two. It ran on Windows and on Mac OS X, and there is a Linux port. There does not seem to be a public release of the code.



ZZZ, displaying a ring rank along the z-axis

RZZ

RZZ was written by Rob Smith. It was the second implementation to support projecting a ZZStructure in three dimensions, and the first implementation to be fully modular. While it is written in C++, most code using it is written in python. It is notable for introducing the Synapse optimization, and for integrating microversioning support.

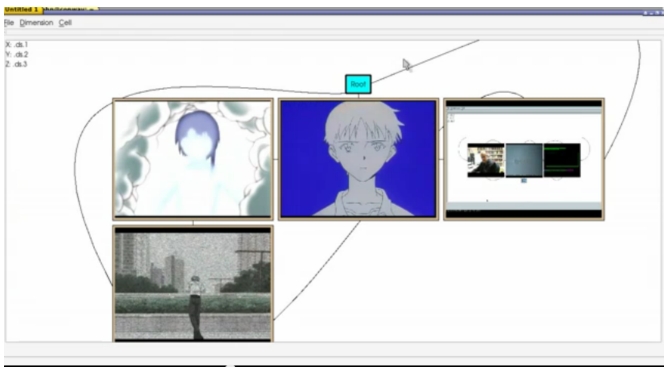
While a public release of the front end does not exist, the RZZ backend is used in several other Project Xanadu projects, including XanaduSpace, demonstrated here: http://www.youtube.com/watch?v=En_2T7KH6RA

A novelty view in an early version of RZZ's front end

Dimscape

While Dimscape does not support KBLANG, it is an interesting and featureful ZZStructure navigator. It was written by Jonathan Kopetz in 2011. It supports multiple cell types, including program cells and video cells. It runs on Windows and Linux.

It is available from here: https://github.com/ccs4ever/Dimscape/tree/master/dimscape



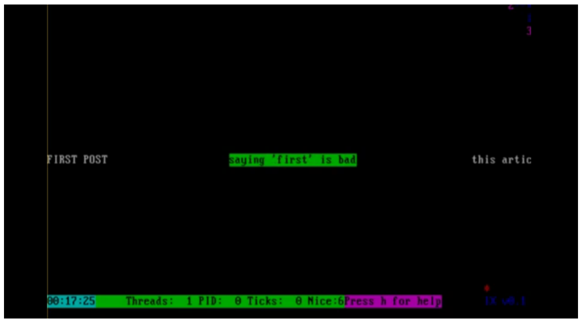
Dimscape, displaying several videos simultaneously while a user navigates between them

Ix

Ix is another ZZStructure navigator. It does not support KBLANG, nor does it support named dimensions or clones. It is notable for being completely self-hosting (it is an operating system), and for having an in-memory and on-disk storage format designed entirely around fast lookup in

ZZStructures. It supports a limited form of grid view called an x-bar view.

It is available from: https://github.com/enkiv2/ix



```
FIRST POST                    saying 'first' is bad                  this artic

00:17:25     Threads:  1 PID:  0 Ticks:  0 Nice:6Press h for help
```

A part of the Ix demo video

Appendix C: Glossary

Cell:	a container for a chunk of information, connected to other cells upon zero or more dimensions
Dimension:	a categorization for a connection
Rank:	a set of cells connected together along a given dimension
View:	a set of input operations and drawing rules for representing and navigating a zzstructure
Head:	in a rank, the cell that has no negward link
Tail:	in a rank, the cell that has no posward link
Posward:	roughly, forward in a rank
Negward:	roughly, backward in a rank
Cursorplex:	a set of accursed cells and display dimensions
Accursed cell:	the cell upon which a cursor rests
KBLANG:	a set of keyboard hotkeys used in all ZigZag implementations, capable of navigating most views
Ring rank:	a rank containing neither head nor tail; going in either direction results in a loop
Clone:	a cell along the clone dimension that is not the head; a clone inherits its value from the head of the clone rank.
Autovivification:	a feature of some interpreted languages, wherein an identifier is created and given a reasonable default value the first time it is used, so that it does not need to be declared or defined before use; this behavior exists in awk and perl

Appendix D: Extra resources

Official website: http://xanadu.com/zigzag/
Adam Moore demo: http://www.youtube.com/watch?v=si1EJ584foA
Ted Nelson on ZigZag structures (demo): http://www.youtube.com/watch?v=WEj9vqVvHPc
A ZigZag-centric design, “Floating World”: http://xanadu.com/zigzag/fw99/

Appendix E: Applications

Databases
A ZZStructure can be used as a non-tabular relational database. Each dimension is the equivalent of a column, and each cell is the equivalent of a row -- an eversion of the tabular model. In this way, we can directly represent organic connections between instances of things.

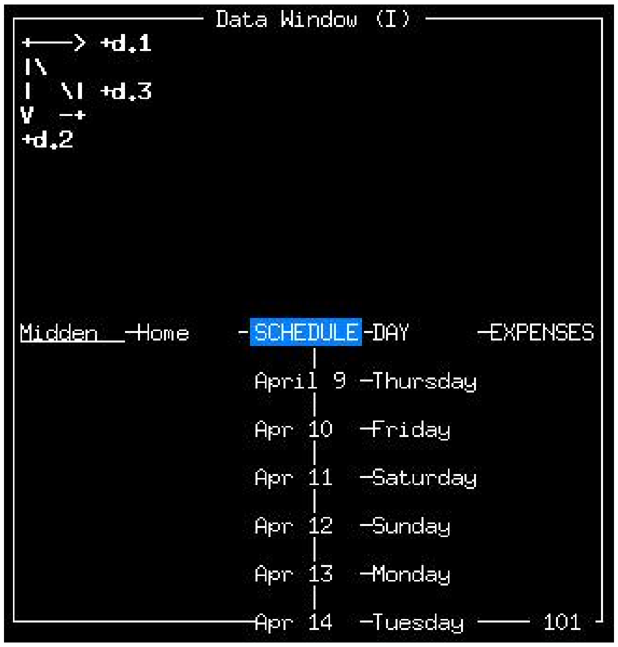
Keys are no longer necessary, because every cell is unique and acts as a key along every dimension upon which it exists. Where SQL rows are differentiated by having unique sequences of values (and have synthetic values added in order to make them unique if they are not), in ZigZag® every cell is unique regardless of its value, and the arrangement of connections can mimic reality in arbitrarily nuanced ways rather than being limited by restrictions based on categorization.

Where multi-valued columns in a traditional RDBMS necessitate either wasteful duplication (repeating rows that differ only in a single value) or wasteful indirection (using auxiliary tables and adding a column for synthetic foreign keys), in a ZZStructure database extra values can simply be added onto the end of the rank, or attached to a clone.

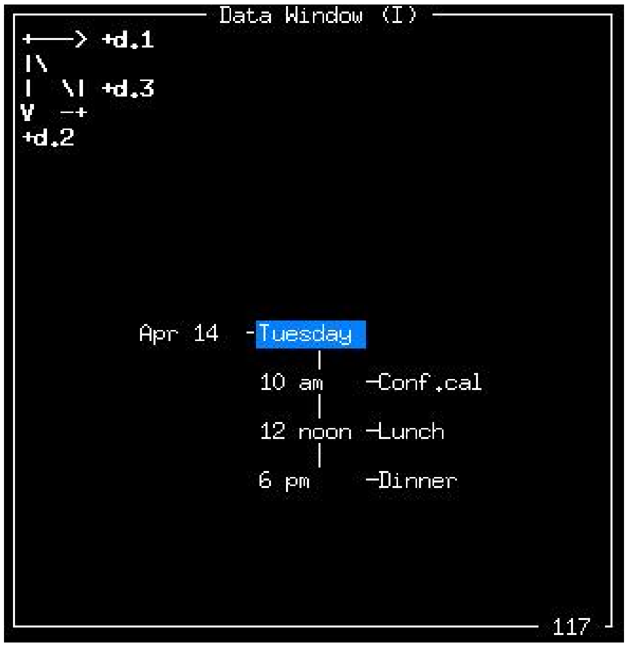
Data visualization
Adam Moore demonstrated the potential for ZigZag to be used in visualizing and manipulating complex data sets with his chem demo (see the GZZ section in appendix B). In it, he demonstrated navigating from the name of a chemical compound to a lewis diagram of that compound, and from there to a periodic table, and then on to a set of lewis diagrams for different compounds containing a particular element. Other kinds of data can be visualized similarly.

Organization
Ted Nelson demonstrated AZZ at the first Wearable Computer Conference in 1998, in a talk called “What’s On My Mind”, showing how it could be used as a personal organizer: http://www.xanadu.com.au/ted/zigzag/xybrap.html

A portion of the demo from this talk is reproduced below, wherein he demonstrates using this software to manage his schedule, finances, and contact information -- replacing the work of a calendar, spreadsheet, and rolodex with exactly one piece of general-purpose software with exactly the same interface, the same display mechanisms, and the same keybindings.

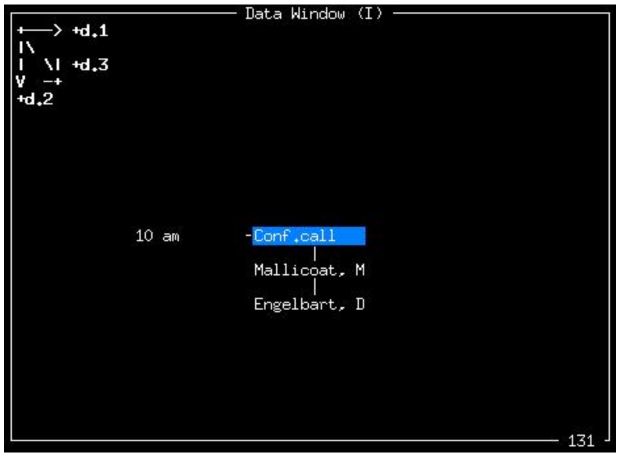


```
                 Data Window (I)
+--> +d.1
|\
| \| +d.3
V  -+
+d.2





Midden  -Home    -SCHEDULE-DAY      -EXPENSES
                  |
                 April 9 -Thursday
                  |
                 Apr 10  -Friday
                  |
                 Apr 11  -Saturday
                  |
                 Apr 12  -Sunday
                  |
                 Apr 13  -Monday
                  |
                 Apr 14  -Tuesday ---- 101
```

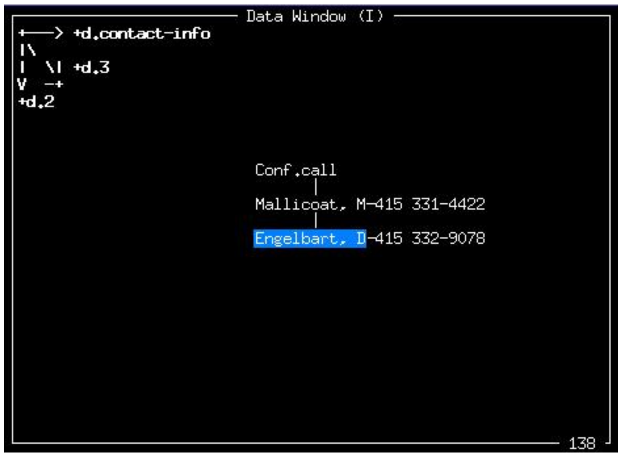


```
                 Data Window (I)
+--> +d.1
|\
| \| +d.3
V  -+
+d.2





           Apr 14  -Tuesday
                     |
                    10 am    -Conf.cal
                     |
                    12 noon -Lunch
                     |
                    6 pm     -Dinner



                                            117
```

Navigating from the home screen to a schedule, and then down to a particular day's schedule

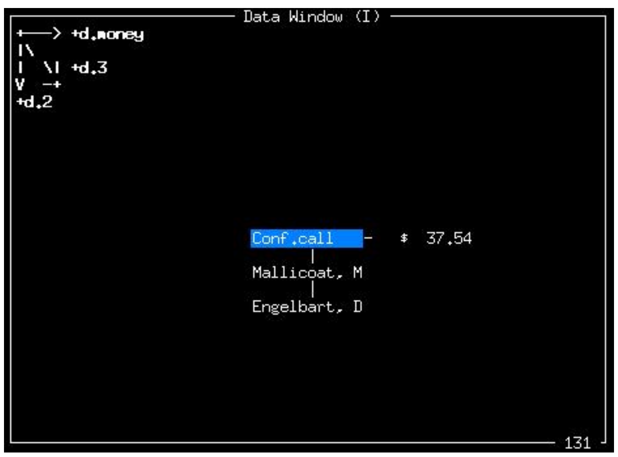


```
                 Data Window (I)
+--> +d.1
|\
| \| +d.3
V  -+
+d.2



        10 am       -Conf.call
                       |
                     Mallicoat, M
                       |
                     Engelbart, D




                                            131
```

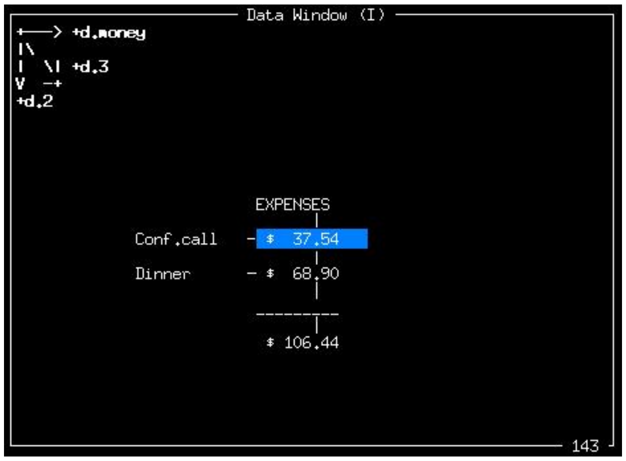


```
                 Data Window (I)
+--> +d.contact-info
|\
| \| +d.3
V  -+
+d.2


                 Conf.call
                   |
                 Mallicoat, M-415 331-4422
                   |
                 Engelbart, D-415 332-9078





                                            138
```

Navigating to a schedule item, and then switching dimensions on a person's name to reveal more information.



```
                 Data Window (I)
+--> +d.money
|\
| \| +d.3
V  -+
+d.2



                     Conf.call  -   $ 37.54
                       |
                     Mallicoat, M
                       |
                     Engelbart, D




                                            131
```



```
                 Data Window (I)
+--> +d.money
|\
| \| +d.3
V  -+
+d.2


                      EXPENSES
                         |
           Conf.call  - $  37.54
                         |
           Dinner     - $  68.90
                         |
                      ---------
                         |
                       $ 106.44



                                            143
```

We change dimensions again and look at expenses