WHOEVER GETS A XEROX OF THIS, PLEASE SEND ME A POSTCARD SO INFORMING ME.
→ T. Nelson, 458 W. 20, New York, N.Y. 10011.
PLEASE DO NOT XEROX AFTER 1968. Write instead to me for most recent intelligence.

HIN

Hypertext Implementation Notes. Theodor H. Nelson.

6-10 March 1968.

Not for publication. Very informal. Cite as "personal communication," if at all.

This is a rough enumeration of all the problems that have been on my mind in implementing hypertexts (problems, not applications). The effort here has been to be comprehensive rather than comprehensible. Many things may be unclear. Others have not been specified ~~etc~~ usefully. Other things have been omitted, presumably.

These notes are an attempt to clarify:

1) Basic examples of hypertexts, in greater detail than elsewhere — Types and mechanisms.

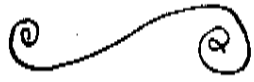
2) The attempted generalities that have kept coming up, causing some confusion.

These things are all being presented candidly here, in the hope of getting across exactly what has been on my mind so that the appropriate implementation details can be handled by ~~those that~~ them as may thenceforward understand them best.

This has all been written cold turkey (without notes or revision), so some sections modify earlier ones. Cross-referencing ameliorates this. But these documents, if understood whole, will pass on the border of seeking overall structures. Perhaps there are none useful:

This is a jigsaw puzzle. Unfortunately, whether or not it makes an overall picture — that is, any unified structure — is unknown.

CONTENTS



This is in no order at all, except that "Types of Hypertext," p(3), might serve as some sort of an orientation. The latter three sections are independent.

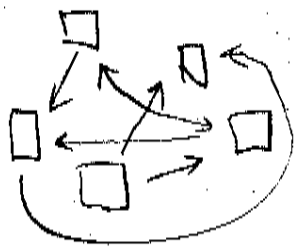
The only thing which approaches a decent level of specification is "1-DIMENSIONAL CONTINUOUS HYPERTEXT," 14 ff. But the different notes cast shadows on one another, and nothing could be implemented except by someone who understood all this [illegible]

TYPES OF HYPERTEXT

The following types of hypertext (categorized structurally and not editorially) are known to me:

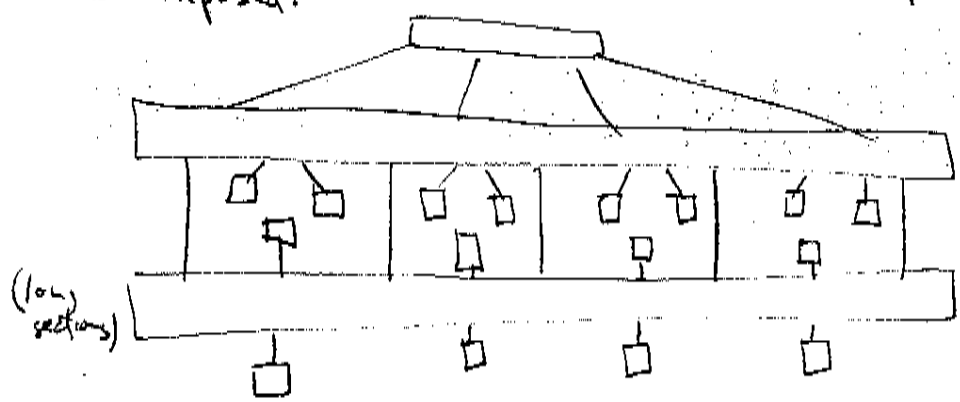
DISCRETE IRREGULAR HYPERTEXT

Individual text sections or chunks joined in a graph structure (one-way arcs or two-way chords).



A choice, usually visible, lets reader pick the next, though it can be a factual question or a default "reading" of sections.

DISCRETE REGULAR HYPERTEXT where some repetitive structure is imposed.



CONTINUOUS HYPERTEXT, where some attribute(s) of the text may be changed by "continuous" degrees (very small increments).

1-DIMENSIONAL. The simplest example is Stretchtext, where the attribute that can be changed is length. But it could be any other attribute, like Humor.

N-DIMENSIONAL. Separate 'throttles' or whatever vary the text's properties separately.

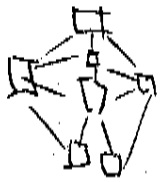
Consider also the following complex texts:

THE PROUSTIAN MANUSCRIPT, with a) indexes, b) cross-reference jumping, c) alternative versions.

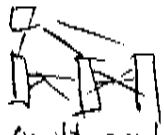
THE HYPER-MANUSCRIPT, same as above but with ~~fuzzy~~ alternative hypertexts permissible.

GRAPH DISPLAY

Virtually essential for hypertext construction and complex text editing is a screen display to show (and modify) graph structure. This includes:

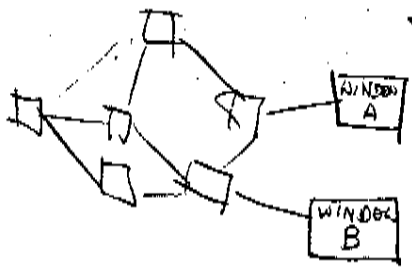


PARTS GRAPHS, to show what there is in a given corpus and how it is interconnected.



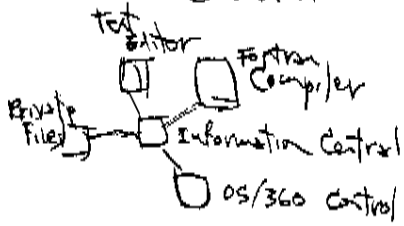
(multicouple)

This is essential for frontier editing and the hyper-library.
Naturally, the amount and types of info displayed at any one moment would have to be variable under user control.
How input would be selected & modified (from equivalent graphs) is an open question.



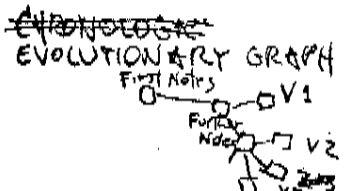
This would also be an essential display for discrete hypertexts — particularly if the 'variable windows' idea of XANADU (see 'XANADU') is implemented.

FACILITIES GRAPH



This is another XANADU idea. (See 'XANADU.') Very important if the hypertext facility is to be linked up to other work, e.g., computer programs.

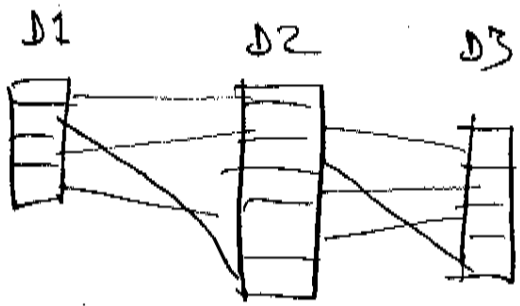
~~CHRONOLOGIC~~ EVOLUTIONARY GRAPH



(For 'Frontier Text Editing' and 'Hyper-Manuscripts' which see.)

The ELF (& previous unification)

In an earlier paper ("A File Structure for the Complex, the Changing and the Indeterminate") I described a file structure thought to be of general use. The idea was to store documents and text structures with linkages among their sections which would not change if we changed the sequence of a particular one.



This 1-to-1 relation among user-selected sections may be used to keep track of various changes among versions of a document and corresponding parts of different documents. This latter may be used for creating tables of contents. Thus it is a rather useful and important relation in this problem area.

The ELF (Evolutionary List File) was to be a file structure which incorporated this relation among ~~the~~ documents filed, and maintained the relationship through changes. PRIDE was to be the larger language that permitted the changes, plus helping housekeeping.

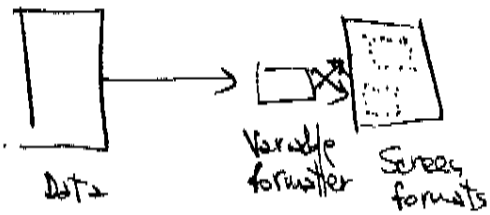
Neither of these terms is useful right now. But we should consider the connector bundle among documents or other data as an important sort of thing. In these papers we will call it simply a "multicoupler." It links user-specified sections, possibly assuming paragraphs to be the sections as a default assumption, subject to user correction.

Multicouplers may be transitive or nontransitive, hereditary through changes or deteriorating, depending on properties the user needs. A variety of types [illegible]

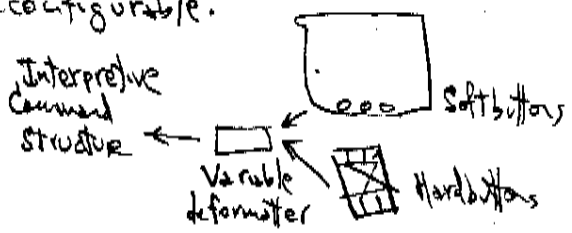
XANADU (a previous unfraction)

The Xanadu system was a half-specified setup worked on at Harcourt, Brace & World. It had several interesting and useful aspects:

1) Screen formats for any kind of work were to be quickly reconfigurable. That is, the user could designate the size and shape of 'windows' into data, and their positions on the screen, thus creating changeable working formats.



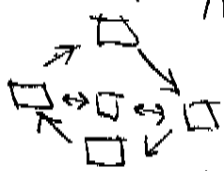
In much the same way, the use of both virtual pushbuttons on the screen, and quick-setup hardware pushbuttons, were to be reconfigurable.



2) Particular data structures — that is, relatively simple ones, like business forms and manuscript pages — were to be easily creatable and stored in a common data-base format. The general intent was to experiment with low-level text-editing and business-information systems.

3) Two types of graph structure were to be ~~sig~~ significant to system behavior. One was to be a graph among text sections — i.e., a discrete hypertext.

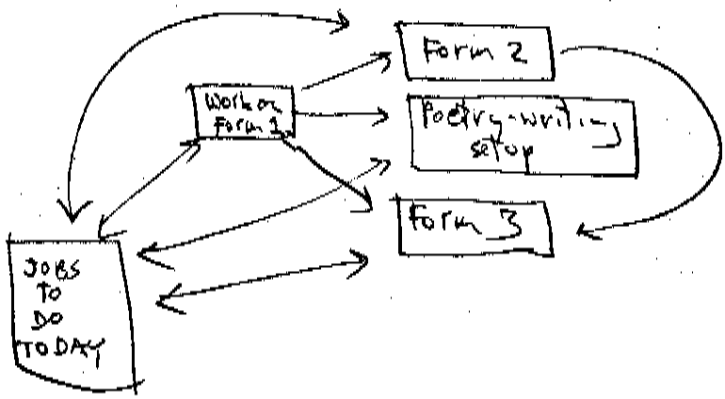
E.g.



The user, setting up his 'windows' any way he wanted, could jump around it by the arrows,

with each window looking into the hypertext at a different place, if he wanted.

The second graph structure was to be the set of activities in the user's "workspace."



Each of these task setups would offer options to switch into the connected task setups. Thus you could work at it all day, doing everything on it, supposedly.

The two graph-structure systems were supposed to share internal formats, and also to be displayable, as graphs, on the screen. (See section on 'Graph Displays.')

POIGNANT (a previous unification)

The incredible jumble of activities, scraps, pointers, and pieces of string to save in those various systems naturally pressed me to think of some fairly general way to handle it all. This was done from 1965 on, taking shape gradually (with the XANADU plan) in a file structure called POIGNANT (because it was mainly concerned with pointers).

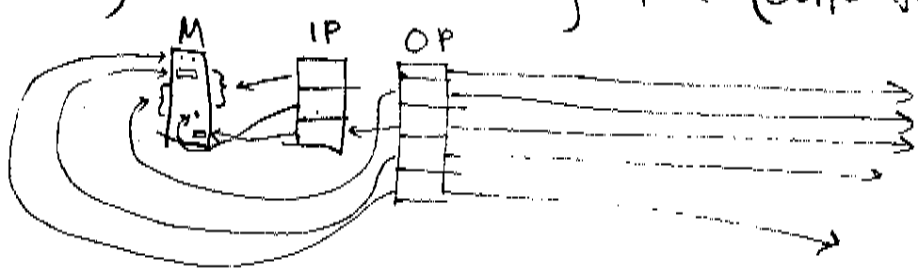
Everything had to be capable of being indefinitely long if necessary. However, it was sort of mostly to be divided into big units made up of little ones

A great variety of pointers, and acknowledgement between them (see 'Hyper-Manuscripts' and Hyper-Libraries) had to be possible.

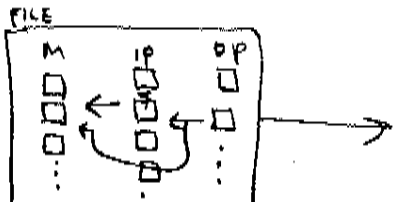
Thus it was decided to have everything in ~~or threaded~~ sections of standardized length, threaded to great possible lengths into "trains" of

This wholly ignored the problem of fast lookup, which meant nothing to me at the time. Not that I was unaware of it, but I had (provisionally) in mind corpuses of sizes where this would not be too painful.

It was also decided to separate three different types of item: 1) text, or better matter (like it could be numerical data); 2) pointers into that matter ("inpointers") and 3) pointers elsewhere, including those connecting something inside with something outside. (Outpointers.)

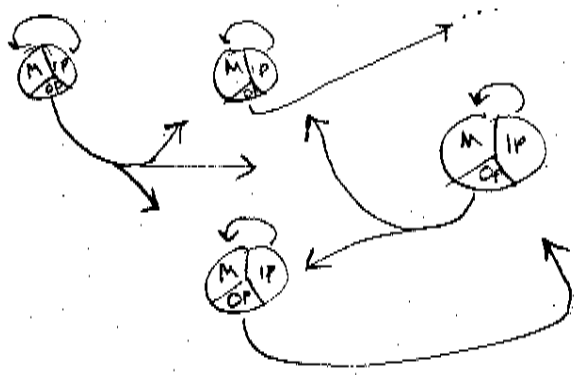


This meant: ~~Since each~~



each of the three types of information was to have its own 'train' within the overall file assigned to some particular thing.

In other words, anything in POIGNANT was to be stored in complex ways that would always reduce (though sometimes vestigially), to



~~In recent months a few ideas have been added or clarified.~~

In recent months a few ideas have been added or clarified.

One was what the hell, you could have trains for different purposes all in the same file. if convenient, for instance, if you were accumulating a manuscript, one train of Matter would be the constituent items, and change orders, just as they came in, and another train, obviously part of the same file, would be the updated thing itself, and maybe another train would be screen buffers. So one file could have a lot of different trains. (Pointers, too, might be sorted out into separate trains for diff. purposes.)

Other modifications of this idea have to do with diplomatic relations among files; especially ways that one file can know when it is pointed at so it won't mess things up by allowing itself to be changed without taking account of that pointer. Thus we are concerned with such things as change buffers, acknowledgement pointers, and relocation addresses within a file.

~~However, this whole conceptual scheme is still unsolved~~

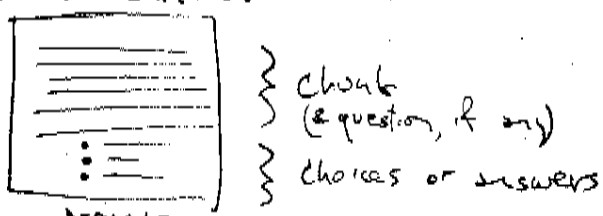
One other concession to feasibility was the idea of an executive record, telling the location of successive records on a train. But a minor amendment to this was the idea that if a train got too long, the executive record would skip and only point to each nth — to keep the executive small & present.

However, this whole conceptual scheme is unsolved, under the present realization that these details can be better worked out by others, once they have the whole picture.

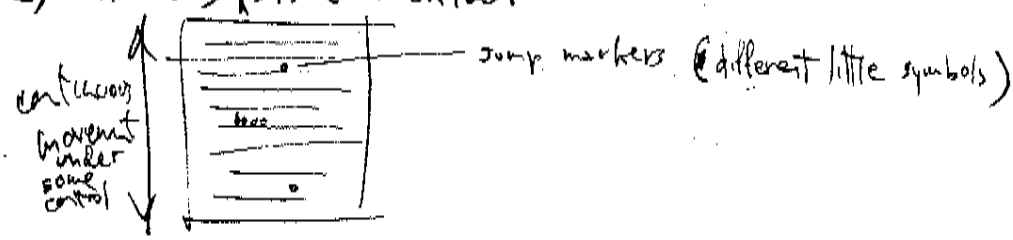
PLAIN DISCRETE HYPERTEXT

Basic screen layout:

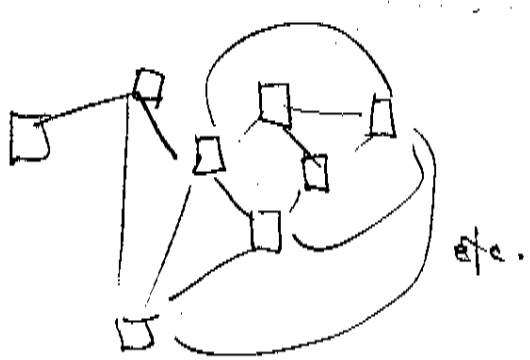
1) JUMPING LAYOUT



2) THREADED DEFAULT SYSTEM LAYOUT



The graph structure for either of the above may be alike:



However, in the case of the jumping layout you ~~set~~ are brought to a stop till you make the next choice, while with the threaded system the text 'looks continuous' — because a Reader Parameter Vector, actually a second graph, determines a complete (or incomplete) set of default options. If you don't choose a jump, you are automatically moved on to a next chunk; and this system of default chunks

+

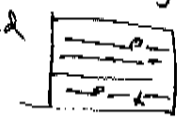
may be varied from Reader to Reader. The different little jump markers of course need to be conventionally established by an author at the beginning of his piece.

Form & storage: Chunks + Graph Structure + ~~Entities~~ Jump Info

(Numbered) ~~Grey~~ Text chunks (say, 128 to ~~8192~~ 8192 characters)

Graph structure with numbers (representing text chunks) pointing to other numbers (representing other text chunks)

Jump info. Where the significant jump markers go in the text and what they look like.

(Note that   may be structurally the same; each is an arrangement of text with light-pen-sensitive (or mouse-sensitive, etc.) jump markers.

HOW PROCESSED. Obvious.

SCREEN RESPONSE.

You show a ~~thing~~, then when a jump marker is hit, you show another thing.

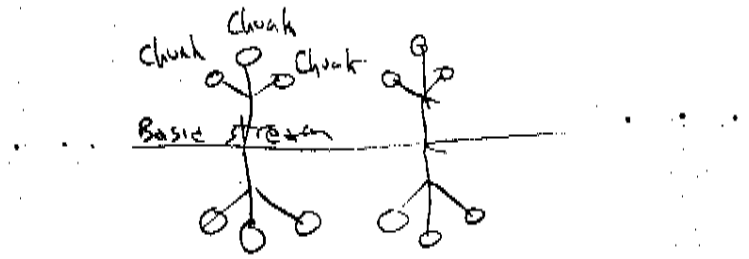
However, continuous up-down movement is also necessary. (i.e., 1/1024 increments)

This is obvious in the threaded-default case; also needed in the jumping case where one chunk doesn't all fit on the screen.

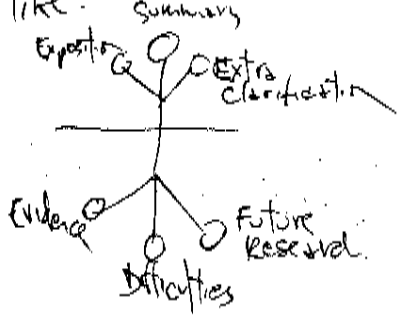
REPETITIVE DISCRETE HYPERTEXT

(this spec modifies Plain Discrete Htext.)

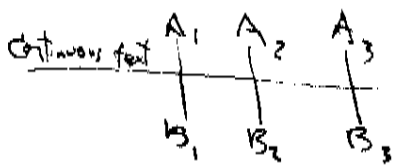
Just like plain discrete hypertext, except you want to have structures that repeat. Polymerize



(This could be like: summary

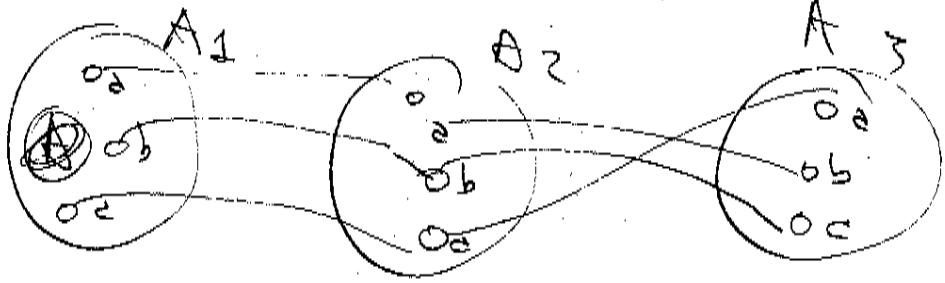


It seems to me that this simply calls for a slight extension of the graph structure system required for Plain Discrete Htext. What must be allowed is the pointing, not just to individual chunks, but to Molecules & positions on them. Thus the above could be represented as



But further, 'molecular structure' would have to be variable

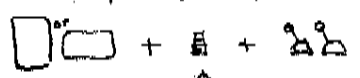
(since you can't expect authors to be consistent) and given positions a, b, c on molecules A must be allowed to interpoint irregularly:



Here we have another "multicoupler (see ELP).
variant type of

1-DIMENSIONAL CONTINUOUS HYPERTEXT
(Especially: Stretchtext.)

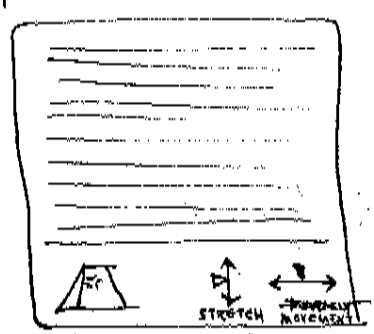
Basic screen layout: screen plus ~~push buttons &~~ mouse, throttles & altimeter.



← Preferably actual throttles, e.g. Lionel dual controller.

Altimeter can be just some sort of scale, or a two-dimensional plot of where you are in the Stretchtext.

Possible 2250 screen layout:



Altimeter

Whether it should be vertical, horizontal or square needs to be determined empirically

Controls on right (right-handed user). Zapping the arrowheads with the light-pen causes requested action for a little while; continuous zap gets continuous movement. (May want some latching button for "keep moving screen both on Stretch & Movement.

Or: a Simpen vector whose length & direction indicate desired stretch & movement. Move around and back to location point to stop all motion.

Form of storage: Depends on strategy.

Strategy 1: (theoretical alternative) store as components, reassemble from surface to bottom.

Strategy 2: Store as tree: finished sections which are then revised on basis of change orders.

Under Strategy 2 the following data structure is required:

ATTITUDE M.N
(a two-part number)

CHARACTERISTIC ⟶ section M
is a chunk of text
(say, 128 to 2048 char.)

MANTISSA ⟶ Change orders # 1 through N
are then applied to section M.

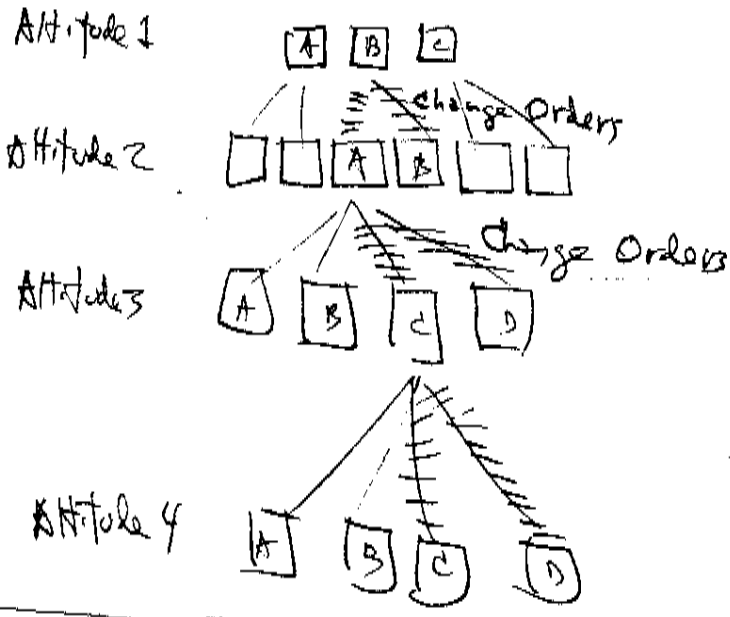
Change orders ~~are of~~ are of three types:

INSERTION + text (1 to 256 char., say)
DELETION + 2 inter-character pointers [beginning & end of thing to be deleted]
SWITCHEROO + 3 inter-character pointers [beginning & end of sections to be switched]

Note that each of these can be ~~created~~ created and undone
operation — the Deletion only if the deleted text is saved
elsewhere, with # in them reflecting pointers to where it came from. Or if text to be deleted
is copied intact into the original
after which it is cut out, etc.

FILE

GENERAL STRUCTURE OF STRETCHTEXT
(Other 1-D Cont. Htexts less clear)

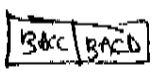


Each file points at those above and below it.

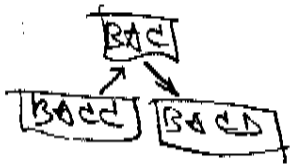
HOW PROCESSED. Let's say reader starts at A, throttles into B. Then he stretches. Change orders ~~are there~~ under B are then applied till he gets to A beneath it (call it BA). ~~Then he keeps stretching~~

7

Reader keeps stretching. From now on system is referencing file BA directly, applying change orders to that. Reader keeps stretching, reaches BAC and BACC (which files now become system referents in turn). Now reader goes forward. BACD is then added to the buffer:

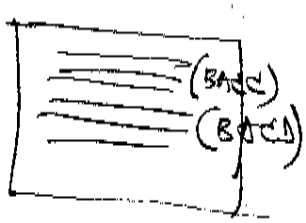


What the system does is a tree-retrest:



Now the Reader shrinks the text. ~~As soon as it~~ The list of change orders is now undone, each being treated as its inverse operation.

Suppose now the reader is at ~~BACC.5~~ altitude 3.5, and halfway between BACC and BACD, thus:



Now he keeps going forward. After BACD he slides into BADA, etc. Each time a record is passed through, the system retrests in the tree to find the next base text section.

SCREEN RESPONSE.

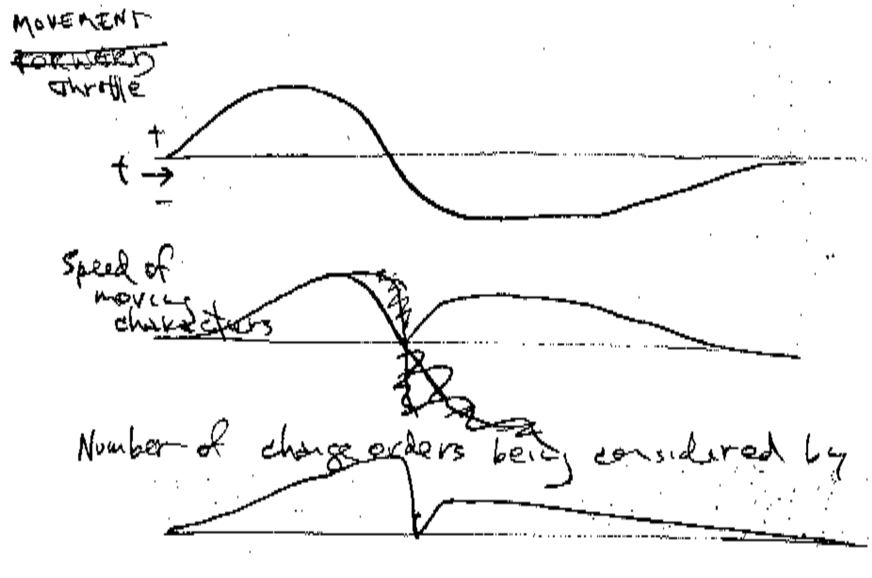
If we had throttles, we could give each a 'neutral' position. As it is, the two controls must be incremental or semi-incremental. However, the movements on the screen should be very nearly continuous. Suppose we had

[text] Somewhat worried, he

[change order] INSERT /after-char 16/about the soldiers/

The ~~boy~~ ", he" should move slowly to the right (& thence to next line) +

Each change order put into effect takes a certain length of time. This should be modifiable under program till we get the timings we like. ~~Example~~ The timing and the combining of the timing should be separately controllable on the basis of "throttle" behavior.



What I am trying to say here is that if you pull hard on the throttle, you should get an overlap (or multiple overlap) of change-order processing; while if you pull softly on the throttle, it processes only a few, or one, at a time.

It should be clear that this exact "feel" is going to be very important, and hence must be experimentally variable — especially if it has to be done by light-pen or even Grafacon.

Probably we need a table (program variable):

Throttle degree	Change order Lookahead/Lookback: number to be simultaneously processed	speed of first c.o. response	speed of #2 resp.	speed of #3 resp.	...

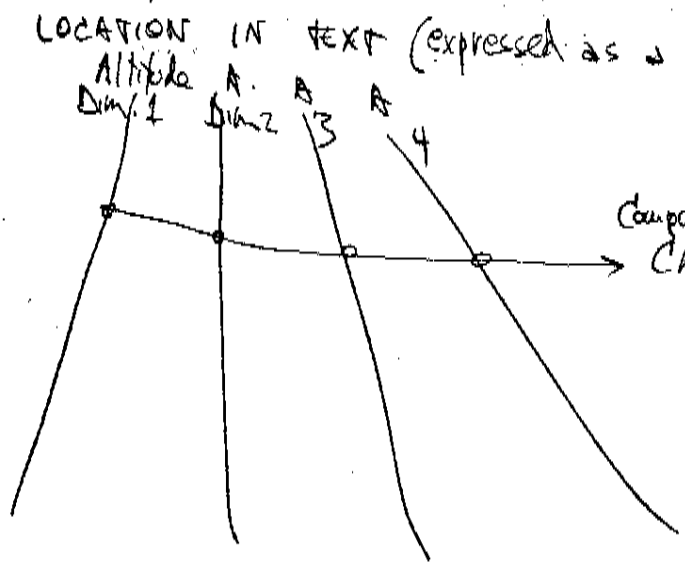
To determine the [illegible] [illegible] [illegible] line of [illegible] [illegible]

we must compute the change orders currently affecting it (from the table), and the relative speeds they impart to it (from the table). As soon as one change order has been finished, or the throttle setting has changed, this must be recomputed.

MULTI-DIMENSIONAL STRETCHTEXT (This spec modifies 1-DCH)

This is essentially the same as the 1-Dimensional Continuous Htext, except that provisions must exist to vary several attributes continuously (small change orders, call them Snaps).

Each attribute may have the same tree structure described for the one-dimensional case, except that each also needs to couple to the others. This would be a function of Altitude for all the different dimensions, then jointly.



[beyond those decreed separately for dif. dimensions] ...
⇨ these are specified by the author in terms of relative altitudes of all dimensions, just as regular change orders are specified in terms of individual dimensions.

How to organize this in core (or disk), I don't know.

PROUSTIAN TEXT EDITING

(This spec Modifies '1-dimensional ~~text~~ text,' esp. by permitting chronological forks in change order system.)

Starting from scratch, you may input your text freely, revise it continuously on the screen, having all your revisions remembered with sequence, time and date; go chronologically backwards through revisions, to see the changes; settling at an earlier point, ~~branch elsewhere~~ begin applying a different set of revisions; declare 'versions' (or other parallel structures, such as indexes) at any chronological point in the revision stream or tree; declare correspondences among the versions and structures; and make annotations of any version, part of version, revision, or correspondence between versions.

Indexes of the manuscript ~~text~~ parts (entry by different versions if desired) may be mouth compiled, also as ~~some~~ structures 'corresponding' to the versions.

Graphs may appear on the screen, reminding the user of all his actions and all the material he now has stored. (See 'Graph Display'.)

SCREEN LAYOUT: Variable. User shd. be able to move windows & modify his Facility graph (ignore for now). (See 'XANADU.')

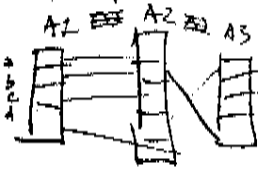
FILE STRUCTURE. Text segments and change orders stored as a chronologically branching graph.

Those may be stored (see '1-Dim. Cont.-Struct') as:

Insertions
Deletions
Switcheroos
AND
forks or branches, numbered.

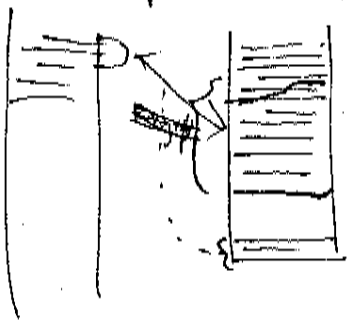
Since a common code should probably be used for this chronologically changed material and stretchtext, this then modifies that:

2) A further 'multicoupler' relation, showing correspondences between discrete units in parallel versions or structures. (see 'ELF' section.)



Note: resemblance to a 'chemical' structure (see Repetitive Discrete Hypertext). Three separate units, here, are coupled with multicoupler A — but the detailed correspondence among sections must be specified to the system by the user.

The Multicoupler as required for this editing system has certain odd properties.



The relation may not be transitive among ~~trib.~~ versions or structures. This is because the elements of one version may become split and dispersed around the version. Hence the multicoupler must be allowed to fork as changes are made in a particular version. Moreover, several different forms of multicoupler behavior under version change must be possible, depending on what the user want ~~(these differences have to do with~~ The 'strict' multicoupler would disconnect from a section once that section was split. ~~The 'tendrilous' multicoupler~~ the 'forky' multicoupler would continue to point t the different parts made from that section. The 'embracing' multicoupler would include everyth between the split sections, unless the interven material was pointed at by another part of the multicou

The 1-for-1 property of the multicoupler should also be relaxed in an optional type of multicoupler.

Specialized sequencing subsystem.

It is desired to allow the user to sequence materials — his own sections, or structure components — on the screen. This may be done by the usual method of inserting or relocating on a scrollable list; or by a method of pairwise comparisons, where the user considers a present item and then says whether it comes before or after certain other items.

It will be noted that a series of such pairwise comparisons is likely to result in an inconsistent overall graph. This is intended. Given a contradictory set of sequencing choices, he must be allowed to undo these choices individually till the overall graph is a satisfactory sequence. We may call this a quasi-sequence facility.

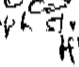
Specialized action pushdown.

The user, when in a wildly inspired mood, must be able to Push Down his current activity and skip elsewhere to do something else, then pop to the dropped activity. This must be possible to a considerable depth.

There must also be a push-away stack, without priority, to which dropped tasks may be relegated for possible return.

Hyper-Manuscripts. Hyper-Libraries.

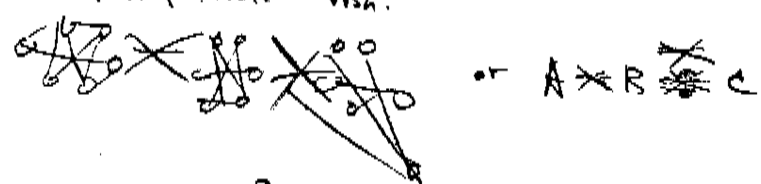
A hyper-manuscript is either an unfinished ordinary text which has been stored in some complex interconnected form (impossible on paper), or a hypertext which is not yet finished and so must be stored in complex forms that include alternatives, undecided. (Continuous hypertexts ignored here.)

This is not particularly different from Proustian Text Editing, except in that it requires treating whole graph-structured texts as the units to be coupled together as 'alternative versions,' indexes of one another, etc. This means that the Parts Graph ~~can~~ must have graphs as its components:  (Map of whole hyperscript, where each blob represents the graph of a component hypertext) ← (See 'chunks!' shown under the 'Discrete Repetitive Hypertexts.' Yet to be recursive molecular notation?)

Therefore the multicoupler must couple to graph structures as well as text sections.

PROUSTIAN MANUSCRIPT

HYPER-MANUSCRIPT



where ⋈ represents the multicoupler relation.

It would seem, then, that the notation for graph structures ought to be recursive, so that a specific relation can be notated as glomming onto a text section or a graph-structured discrete hypertext. The graph structure of a hyper-manuscript includes the graph structures of its component hypertexts.

A hyper-library is a facility which stores hypertexts and makes them available. It has a key problem in common with the hyper-manuscript: if a component hypertext is coupled into (by the writer of another hypertext, or by the student taking notes), this now means that ~~version~~ hypertext cannot be changed.

unless there is also provision to save this version of it (coupling it to later versions as well). (While other rules or arrangements might be made, this is the case we have to think about.) Call this a fixating pointer, and a multicoupler into such a version a fixating multicoupler.

(Whether this fixating means saving a whole version, parts of the version, or reverse change orders permitting its recreation, is a matter of indifference.)

This same problem arises with the hyper-manuscript. A component hypertext must somehow be _informed_ that it has been coupled into, so that this version will be saved (or modified perhaps according to some rules that preserves the desired part of its content).

It is further the case that the user (of hyper-manuscript facility or hypertext library) must be able to create pointers, e.g. for annotation, to any text section, punctuation, type font information, _change orders_, _other pointers_, or other recognizable information within the system. And the 'informing' and 'fixating' mentioned in the above two paragraphs must occur in all these cases.

(Note that we discussed these matters in a very confused way last June: this was the purpose of the various acknowledgment backpointers and 'Wilco bit.')