GZigZag spec

$Id: zzspec.wml,v 1.72 2001/10/08 04:36:43 tjl Exp $

Written by
Tuomas J. Lukka
Benjamin Fallenstein
Antti-Juhani Kaijanaho
(add your name here if you do any significant modification)

You are reading an ugly version of this document that doesn't use CSS to place floats nicely. Figures are shown in a very crude way. Get Mozilla and see the other version - it's MUCH better.

Introduction

The purpose of this document is to be a living specification of the features in the GZigZag system. The GZigZag system is an implementation of the ZigZag structure, invented by Ted Nelson. Much of this document is simply a somewhat more verbose version of discussions with him but other places go into more technical detail.

Some parts of this specification are not yet correctly implemented by the current version, but in these cases, this document is correct (or it **should** be ;-) and the implementation wrong.

Some parts of this spec are at the moment just general ramblings about a topic - once we have the pole editor, I will definitely rearrange it completely. The idea is to try to make them more and more like a true spec as time goes by.

The parts marked with the dreaded triple-X symbol (XXX) are as yet incomplete and should be taken with a not only a grain but a mountain of salt. They mostly contain just a few loose sentences setting the topic.

If you see anything suspicious, feel free to ask me at `lukka@iki.fi`.

The role of ZigZag in the overall scheme of things

ZigZag is a delightfully simple way of operating structures. As such, it has its own uses simply as a personal information manager for people who like

multidimensionality,

In addition to stand-alone use ZigZag is related to Ted Nelson's Xanadu system; a newer version of Xanadu is being designed to make use of ZigZag as a platform. As a brief description of Xanadu, it has

Stable media streams (permascrolls)
: A stable media stream is an invariant, addressable stream of data, e.g.~text, video or audio.

Documents
: In Xanadu, documents are simply lists of spans from the stable media streams.

Content links
: The linking model is fundamentally different from anything else hitherto seen. All links are handled on the lowest level, i.e. as links between lists of spans of stable media streams.

Finding by content
: There are fast ways to find out in which documents a particular point in a stable media stream is included. All content copied via cut&paste can be tracked.

ZigZag can also work as a platform for other types of applications --- or preferably applitudes, meaning that they should also expose the other possibilities of ZigZag *at the same time* as being normal applications. This is so that the interconnectivity provided by ZigZag can be used to make the whole of two applitudes greater than the sum of the parts.

Cells, dimensions and connections

A cell is the fundamental container of data. A cell can "physically" contain either a text string or a single, contiguous span (which is an address, and references a permascroll).

A cell has an ID, which is a string. Currently, if a cell ID contains a colon (:), the cell belongs to a *space part*, where the space part ID is everything from the beginning up to and excluding the colon. If a cell ID contains an at sign (@), it is a *global ID*; everything starting from and excluding the at sign to the end of the ID is the ID of the cell's *creation space*. If the creation space ID differs from the ID of the space the cell is in, the cell is *foreign*, otherwise its *native*.

Cells may be connected to each other along dimensions, so that each cell can be connected to another cell in two directions (negative and positive) on each dimension. The more global structure is not constrained: any two cells can be neighbours on any dimension, but some of these connections are used for interpreting the structure into views and actions (see below).

An important way to describe the space is to take dimensions to be "objects": the dimensions are invertible mappings between cells. This is opposed to the immediate way of thinking "cell and connections", and there is a difference. This view is important because here it is easy to discuss special dimensions that have interesting properties.

Tuukka, I guess your work starts about here.

It is not yet clear whether dimensions are strings or whether they are cells. The various slice and compound space implementations will probably affect and depend on this. If dimensions are cells, then several uniqueness problems are quickly solved, but their being strings may be easier on the users.

One important point is that of headcells: on non-circular ranks, the headcell is the cell at the negative end of the rank. Ted specifies that all ranks should have a headcell and that there should be a way to specify the headcell of a circular rank. The exact mechanisms here are as of yet unclear.

Subspaces, closed cell sets, closures

The following terms may prove useful from time to time:

Closed set of cells
: A set of cells C is **closed** with respect to a set of dimensions D if and only if all neighbours of the cells in the set C along the dimensions in D are also in the set C.

Semiclosed set of cells
: A set of cell C is **semiclosed** with respect to a set of dimension-direction pairs D if and only if all neighbours of the cells in the set C along the dimensions and their associated directions in D are also in the set C.

Closure
: Let us have a set of cells C and a set of dimensions D. A set C' is a **closure** of C with respect to D if and only if it is a minimal closed (with respect to D) set containing all the elements of C.

Semiclosure
: Let us have a set of cells C and a set of dimension-direction pairs D. A set C' is a **semiclosure** of C with respect to D if and only if it is a minimal semiclosed (with respect to D) set containing all the elements of C.

Subspace
: A subset C of the set of cells in a ZigZag space Z is a **subspace** of Z with respect to a set of dimensions D, if and only if C is closed with respect to D in Z.

Proper subspace
: A subspace C of a ZigZag space Z is a **proper subspace** if it is closed with respect to the set of dimensions D which contains every dimension of Z except `d.cellcreation`.

Semisubspace
: A subset C of the set of cells in a ZigZag space Z is a **semisubspace** of Z with respect to a set of dimension-direction pairs D, if and only if C is semiclosed with respect to D in Z.

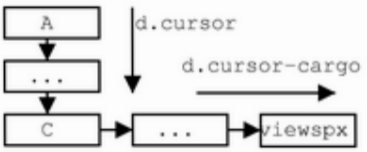
Generated subspace
: A set of cells C in a ZigZag space Z and a set of dimensions D are said to **generate** a subspace C', which is the closure of C with respect to D in Z.

XXX: are these definitions OK?

Key structural mechanisms

Cursors

THIS IS WRONG AND OUT OF DATE.



The path from the viewcell `viewspx` to the cell the cursor of the view points to `A` via the main cursor cell `C` which may or may not be the same as `viewspx`. `C` is at the negend of `d.cursor-cargo` from `viewspx` and the accursed cell `A` is at the negend of `d.cursor`.

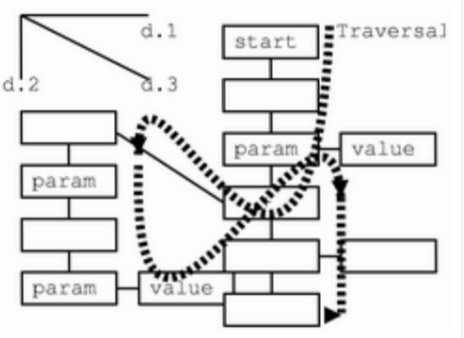
The viewcell points to a particular cell where its cursor lies. The idea of cursors is not restricted to cursors of views but, as we shall see below, anything where selecting a single cell is important. This makes it possible to create a new view to select that cell on the fly (see below).

In the structure, a relcell is used so that several cursors can be maintained pointing to the same cell. The path to find the cell the view is pointing to is start at the viewcell, go to the posend on `d.mycursor` and then to the posend on `d.cursor`.

In order not to disturb other cursors, the insert operation should be used to move the relcell to a new `d.cursor` rank. This operation will leave the original and the new rank otherwise intact. It is possible that any other operation on `d.cursor` will be declared illegal.

Inheritable parameter lists

NOTE: this is liable to change. It is only documented here for understanding.



An example of the inherited parameter traveral to find a parameter value. Starting from `start`, we go poswards on `d.2`

until we reach a cell with a negward connection on `d.3`. We include that list and return to the main list. The parameters that have values store them on `d.1` or as cursors starting on the parameter cell.

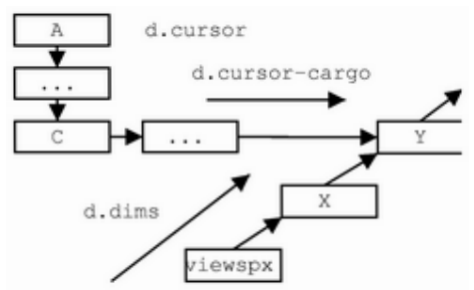
This section defines a way to inherit parameter lists based on the structure.

Views

A view is represented by one maincell in the structure, called the viewcell. This cell can contain text which will be interpreted as the ``title'' of the view, but this is not mandatory.

XXX Since some views which contain more than one cursor are planned, it is possible that something will change below.

Visible Dimensions



The path from the viewcell to the cell specifying the Y dimension shown. First, we go two steps on `d.dims`, then use the same method as in the previous figure for getting to the accursed cell from there. Note that both the `viewspx` cell and the `x` have cursors similarly attached to them.

Dimensions that are currently visible are listed on `d.dims` from the viewcell, in order X, Y, Z (and possibly others, in case of complicated view rasters). The dimension cells are treated as cursors, i.e. the cell representing the dimension is found by going to the posend on `d.mycursor` and then to the posend on `d.cursor` from the dimension cell.

The point of this odd-seeming arrangement is, as alluded to in the Cursors

section above, is that we can very simply create a new view that is bound to a dimension cell of another view.

Dimension lists

In order to change dimensions shown, the views provide an operation ``move the X/Y/Z-dimension-selector-cursor one step pos/negwards on `d.2`". The usual plain-vanilla dimension lists are thus simply lists of cells on `d.2` which contain the dimension name as a string (possibly by cloning). Most often these lists are cyclic but this is not mandatory: trying to move past the end/beginning of a list simply does nothing.

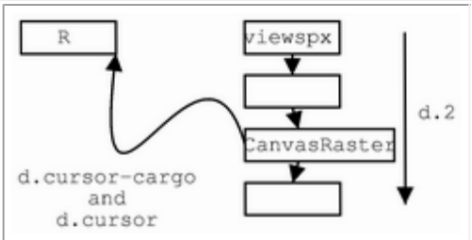
This is just the default arrangement - the user is free to create other dimlist operators and use them to change the dimensions taking advantage of the structure in a different way. One example of an operation we may want to provide at some point is "change dimension but not to one already being shown".

To select the dimension list for a particular dimension of a particular view, the user can simply create a new view for the dimension cursor, using the cursor-cargo mechanism. To set all the dimensions to the same list, the user can set the X dimension and then invoke a special operation that sets Y to the `d.1` poswards neighbour of X, Z to the neighbour of Y and so on, on the same list.

Raster definition

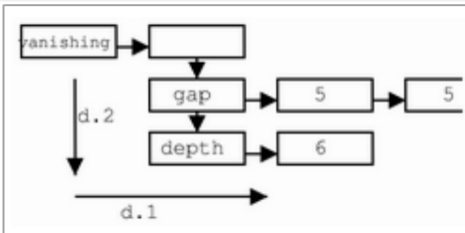
Vanishing raster

XXX



Getting to the raster-defining cell `R` from the view specs. Note that the cursor-traversing operation is abbreviated to the curvy arrow. Also note that the inheriting mechanism defined previously is allowed.

The raster to use is obtained similar to the dimensions to use; only the first motion is different: down on `d.2` until the text `FlobRaster` is found.



A sample specification of a

vanishing raster, specifying values for two parameters. Here, the `vanishing` cell would stand in for the `R` cell in the previous image.

The raster itself is specified using a corner list. An incomplete list of parameters follows:

shrink
: 2 floats; the numbers to multiply the width and height of the cells with.

gap
: 2 ints: the number of pixels of space to leave between cells

depth
: 1 int: The number of cell "layers" to draw.

dimorder
: 2.. ints. The precedence of dimensions.

initmul
: 1 float. If the default size of cell will be multiplied by a number for the center cell (it is usually good to make the center cell larger

There are currently some undocumented interactions to provide for hard rasters; see the source.

Simple Flob Raster

The simple flob raster displays flobs extracted from the structure, on dimensions given by the structure. The dimensions are given by attaching paths to the cells in the dimlist.

Primitive operations

Connect

Insert (XXX rename)

Mono-chug

This operation changes one connection. It takes two directions and one cell as parameters. XXX image.

Bindings

Keybindings offer a nice way of seeing how flexible the ZigZag structure is for programming.

Most fundamentally, the names of the keys (e.g. `Ctrl-k`) are on cells connected along `d.2`. The action for each key is the poscell on `d.1` from those cells - this way several keys can be bound to the same action quite easily.

However, this is not yet sufficient: there may be several input states, e.g. with the vi editor there is the insert mode and the command mode, and inside the command mode there is the special mode of waiting for a motion command. So in addition to the command, there needs to be a way to specify the next state. Likewise, some states are similar to each other so states should be able to inherit commands from each other in various ways.

GZigZag uses blank cells to represent inheritance: when the routine that searches for a key is going down along `d.2`, it will go to the poscell from that cell and check all the bindings on that dimension before returning to continue the search along the original rank.

The cursor mechanism is useful for keybindings as well: the current state can easily be maintaned by a cursor as shown above, allowing the user to easily set the cursor to a different state. However, care has to be taken here as moving the cursor in question can render keybindings unusable, so it is better to use an interface where the cursor can be immediately transported to the right place (such as clicking with the mouse).

The actual structure is currently such that the bindings cursor is the posend on `d.bind` of the view cell. This cursor points to the place where the inheritable parameter search for the next binding is begun. The cursor is set, if the inheritable parameter's value (on `d.1`) has a negward connection on `d.3` -- it is set to the negend.

Additional problems arise when the same view can show different rasters. A different visualization of a structure often makes different means of navigation necessary, or different operations desirable. To provide for this, it is possible to assign a raster a list of modifications to a bindings mode; when the mode is selected, the raster's modifications are searched first, and the standard bindings for the mode are searched when the binding is not found in the modifications list.

A raster can be assigned two groups of modification lists: one to be applied when the raster is selected in the left (control) view, and one to be applied when the raster is selected in the right (data) view. These groups are obtained from the same corner list as the parameters given to the raster (see above); the text searched for is `ctrlbindings` and `databindings`, respective.

From that cell, an intersection on `d.1` and `d.clone` with the current bindings mode is searched. That means that a mode which is intended to be modified is cloned, and attatched to the raster's bindings cell on `d.1`. From this cell, a definition for the binding is searched on `d.2`; if none is found, the mode's default bindings are invoked. For any of these searches, the standard parameter inheritance (see above) is allowed.

This method is practical because it "feels" very much like the definition of the modes themselves, as the modes are commonly listed on a `d.1` rank from the `Bindings` cell on the system list. On the other hand, it's somewhat different structurally.

Default bindings for the orthogonal rasters

`$Id: keybindings.wml,v 1.4 2001/03/02 07:13:37 ajk Exp $`

These are still being discussed and are not yet frozen. Note that there is a panic button: if there is no binding for `ESC`, it invokes undo.

Key(s)	Binding
Ctrl-`S`	Commit the current changes to disk.
`ijl,kK`	Up-left-right-down-Z+-Z- in view 1.
`esfcdD`	Up-left-right-down-Z+-Z- in view 0.
`n` *dir*	Create a new cell in given direction
`b` *dir*	Break a connection in given direction
`h` *dir*	Hop the accursed cell in the given direction. Hopping means simply switching the places of the accursed and the indicated cells in that rank, no other ranks are affected by the operation.
`xyz`	Show next dimension on list on X/Y/Z in view 1
Alt-`xyz`	Show previous dimension on list on X/Y/Z in view 1
`XYZ`	Show next dimension on list on X/Y/Z in view 0
Alt-Shift-`XYZ`	Show previous dimension on list on X/Y/Z in view 0
`/` *dir*	Connect the center cells of the right and left views in the given direction, if no connections will be broken by this.
`~`	Exchange the cursor positions of the two views (no other view parameters are changed).
Delete	Delete the center cell of view 1 and move the cursor. Cursor move tries following directions in order: X-, X+, Y-, Y+, Z-, Z+ and finally goes to home cell.
`m`	Mark or unmark the cell in view 1
Enter	Execute cell in view 0, with the cell in view 1 as parameter
`v`	Change raster in view 1.
`V`	Change raster in view 0.
Alt-`v`	Change raster in view 1 backwards.
Alt-`V`	Change raster in view 0 backwards.
Home	Go home: move view 1 cursor to homecell.
Esc	Move both views to home and reset dimensions to first three dimensions on first dimlist.
`0123456789`	Insert the given number into the number insert buffer for cell IDs.
`g`	Move view 1 to cell whose ID number was in buffer
`G`	Move view 0 to cell whose ID number was in buffer
Backspace	Remove one number from the number insert buffer
`t` *dir*	Clone the view 1 cursor to given direction (may be in either view).
`T` *dir*	Clone the view 0 cursor to given direction (may be in either view).
`%`	Exchange the X and Y connections of the two accursed cells with each other.
`o`	Go to original (rootclone, cell from which accursed cell was cloned)

	in view 1.
o	Go to original (rootclone, cell from which accursed cell was cloned) in view 0.
End *dir*	Move to te end of the rank in the given direction.
<	Set the cursor of the left-hand-view to the cell the right-hand view is pointing to.
>	Set the cursor of the right-hand-view to the cell the left-hand view is pointing to.
- *dir*	Connect the current view1 cursor into marked cells in given direction. Direction must be in view 1.
Alt-c	Switch into "curseling" (cursor selection) mode (see below).
a *dir dir*	Monochug: change one connection. See above.

Text edit mode

Key(s)	Binding
Esc or Tab	Switch off text edit mode
Delete	Delete one character after cursor
Backspace	Delete one character before cursor
Left, Right	Move the cursor within the current cell
Home	Move the cursor to the beginning of the text in the current cell
End	Move the cursor to the end of the text in the current cell
Ctrl-A	Move the cursor to the beginning of the current line in the text of the current cell
Ctrl-E	Move the cursor to the end of the current line in the text of the current cell
Enter	Insert a line separator in the text before the cursor
any key producing a printable character	Insert the character in the text before the cursor

Curseling (Cursor selection)

As an intermediate for multiple cursors, there is a key binding mode for "curseling:" cursor selection. In the standard keybindings, Alt-c is used to go into this mode, which by default supports the following key bindings:

Key(s)	Binding
Tab, Spacebar, Alt-c or Esc	Select this cursor, quit curseling mode.
Left, Right, jl	Select next/last cursor in system cursor list in view 1.

Up, Down, `i,`	Select next/last cursor positioned on the same cell in view 1.
`sfec`	Like Left/Right/Up/Down, operating on view 0.

Note that you can create four additional cursors through executing the CreateCursors command, found in the action list, by centering the left view on it and hitting Enter.

Notification, events and synchronization

XXX

There are many situations where one may want to be informed about changes to the structure, either before it happens (with the possibility of vetoing it), or after (e.g.~to reraster a window).

Updating views

Out of an email from Ted,

```
There are several issues here.  One is just the mechanics
 of a clean method of synchronization.  The other is
 the problem of A FEELING OF FAST RESPONSE--
 which means, very importantly, PREVENTING the
 non-current windows from refreshing in order to have
 truly INSTANTANEOUS response to each user action.
 Anyway, as much as possible.
```

In order to ensure a speedy response, we have to consider the numbers. There are thousands of cells. Each view shows some fraction of them. At each time, there are probably 1-10 views open and changes to cells will reflect in several of them.

Therefore, it is probably unnecessarily slow to store the information about which cells are seen by which views (except if it is naturally stored by the view itself). Rather, for views, there should just be a global list which gives the priorities with which the views get to refresh themselves.

Now, it makes no sense to start updating at each change to the structure - internally there has to be some method to freeze and thaw the global update

queue.

One interesting problem here is finding which view is really the current view: for instance, in the two-part view Also, displaying only the central portion of the views whose cursor changes first could make a big difference. Or the rank along which the cursor moves plus some small number of cells.

Internal triggers

However, this is not the whole story: views are not the only things that need to be informed when something changes. It should be possible to define per-cell triggers as well.

This is possible in the Java code by passing an extra parameter, a `ZZObs` to a routine that returns some structural information, such as a neighbour, a headcell or the contents. The ZZObs will then be called *once* after any of the items it is observing changes, i.e. the return value of the function the `ZZObs` was passed to has changed.

The callback is not necessarily instantaneous; rather, all the callbacks are grouped and run after the activity that caused the trigger has finished.

Cursor triggers

Eventually, it should be possible to define these triggers in the structure as well, but the best mechanism for that is not yet known. However, for some applications these can't wait. Especially important are cursor triggers, which do something whenever a cursor changes value.

A cursor trigger is placed by placing the action (XXX In what form) on `d..cursor-trigger` of the cursor-cargo cell. These actions are queued to be called sometime after the cursor is changed (usually *before* the next screen update).

Cell types

Cell ID types

The most common cell type is a plain cell. The id of a plain cell is in the format

mediaserverBlock-localId

where *mediaserverBlock* is the mediaserver block containing the space version that created that cell, and *localId* is the id of that cell inside *mediaserverBlock*. This is unique **only** among the cells created in the same *mediaserverBlock*.

Transcluded cells have the format

tid;origId

where *tid* is the id of the cell identifying this transclusion, and *origId* is the id in the mediaserver block transcluded from.

Cells that transclude text spans to the vstream dimension have the format

tid;*mediaserverBlock*$*offset*

where *mediaserverBlock* is the mediaserver id of the scrollblock containing the transcluded span unit (e.g., character), and *offset* is the offset in natural units inside the scrollblock of the unit represented by the cell.

Cells that are in slices have the format

sliceId:*origId*

that is, the same as transcluded cells except that the separator is the semicolon instead of colon.

Slice-included cells cannot be transcluded, and transcluded cells cannot be used as base cells for slices so an Id like

sliceId:*tid*;*origId*

means a transclusion included from a slice, and

tid;*sliceId*:*origId*

is impossible.

Transient cells

Most of the structure is persistent but some parts should be transient in order to save space and time. For example, if the information about the rastered representation of a part of the space (coordinates on the 2-D display) is stored in the space, this would waste an enormous amount of space each time the cursor moved.

The transiency is specified at the time of the cell creation on the Java level. How this figures in the space is yet to be decided.

The transient cells are not trivial: because of design considerations, they can't simply override connections and have all their connections appear deleted in the persistent space. Instead, the persistent space should appear as if the cells had been truly deleted using the delete operation, which causes the cells on opposite sides on a dimension from the cell being deleted to be connected to each other. In other words, if we have the rank ABCDE and the cell D is a transient cell, then this should correspond to a rank ABCE in the persistent space.

Slices

The slice design is what Ted ultimately wants but because it's somewhat more complicated to implement, it's postponed until some other parts of the system clear up (most importantly synchronization) and the existence of other types of

subspaces and their interaction.

Versioning

A key mechanism for versioning is timestamps, which specify moments in the past. The full state of the ZZ space at each timestamp can be accessed through the file formats.

The past versions of cell structures are shown in the structure as virtual, non-modifiable cells, except for the one allowed source of modification: `d.cursor`. The dimension `d.cursor` is a source of much headache in versioning because of its nature but the current solution is to move past versions of `d.cursor` to `d..cursor-past`.

The past versions of a cell can be accessed on `d.version`, and the past versions where the cell's content or connections have changed is on `d.cell-version`, and the past versions where the cell's content has changed are on `d.content-version`. So `d.cell-version` skips on the cells of `d.version`, and `d.content-version` skips on the cells of `d.cell-version`. The skipped-to cells are the *first* cells with the new, changed property.

XXX Can we do `d.cell-version` efficiently???

Stable media streams

Referencing stable media streams is an important part of the overall design. For the Dominica project, we need a subset of the features. The stable media streams in the current design are built on top of mediaserver blocks (see the mediaserver documentation).

The old design and its problems

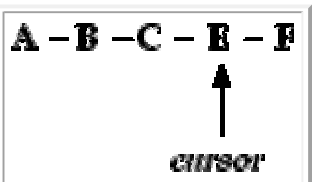
Earlier on, the design was that a cell could directly contain one *span* -- a reference to a continuous range of units from a mediaserver block. (Note that the units can be Unicode characters (text), frames (video) or samples (audio)). A cell could not contain a list of spans directly. It would indirectly refer to a stream of spans by being connected to a rank of other cells, each containing a single span.

This format makes for complex code; Consider what happens when text is inserted into the middle of a span contained by a single cell. The cell needs to be "split" in two, creating a new cell that will take part of the span in the first cell, so that a new cell can be inserted between them. (XXX Image!) All cursors pointing to a position inside the cell (image!) need to be updated. Additionally, there was no easy way to go "n characters forward"-- this always required some complex computations counting characters in cells.

Nile ("Nile Is a Literary Editor"), one of the first ZigZag text editors using spans, was horribly complex and full of bugs, partly due to this inadequacy.

Therefore, there is a new design which avoids some and hides others of these complexities.

The new design



A stream of characters; each character is in its own cell. A cursor points directly to a position in the stream.

The same stream of characters; a new cell with the character "D" has been inserted between characters "C" and "E" (imagine an ordinary `Cell.insert` call has been used). The cursor accursing "E" stays on its position. Note: it has not been moved; it simply wasn't affected by the insertion operation.

In the new design all editing operations *standard ZigZag ops*: insertion and removal are simply normal connect and disconnect operations. Cursors stay on the characters, even if cells are inserted or deleted before their position.

All this is archieved by making a cell on the stream dimension contain only one single unit (character, frame, sample) from a mediaserver block.

Of course this is only the external "API"; internally, we do not create an ordinary cell per character/frame/sample; that would be woefully inefficient. Instead, internally the cells and dimension are represented efficiently by a single object per span. After being created atomically, these spans already form a rank on the media stream dimension. They can be inserted into an exising media stream as a whole.

The most important difference between the old and new designs is that in the old design, the complex code handling e.g. splitting spans when inserting was above

the structure, i.e. all code accessing the structure would need to handle all difficult cases. In the new design, all the difficult cases are handled below the structure, presenting a unified view for the programmer of applitudes.

The internal structure

See "cell types".

Cursors

There are two distinct ways we want to accurse cells in media streams: as cells, or as positions in a stream. Imagine we have a cell accursed in a ZZ view that contains the text "foo." There is a cursor after the f-- "f|oo." Then, we accurse a position in the media stream-- the position between the f and the first o. On the other hand, we might want to look inside the stream, i.e. rotate to the media stream dimension and actually accurse the cell containing the first o. In the first case the cursor points to a *position inside the cell containing "foo"*; in the second case the cursor points to *the cell containing o.* (XXX Image) (Another reason for this distiction, clang pointing, is described below.)

Besides the different meanings of these posititions, they need to be treated differently when cells are removed from the media stream. In the above example, "f|oo," consider the case that the first two characters are deleted, letting only the second o remain. Now, the text should read "|o"-- that is, the pointer should still be in the same stream. (XXX Image) If, on the other hand, the second o was accursed as a cell, the cursor should remain on that o: we want to see where the cell "travels."

Now, consider that a pointer into a media stream is at the same time a cell pointer to the media stream as a whole. That is, the "f|oo" pointer accurses the *position in the stream*, but it indirectly also accurses *the stream as a whole*: to the view layouting the cells on the screen, the "foo" cell (containing the whole stream) is in the middle. Only the code dealing with cells' contents sees the cursor as pointing to the gap between f and o.

The `Cursor` class thus supports the `Cursor.get`, `Cursor.set`, `Cursor.getPosition`, and `Cursor.setPosition` functions (all static). `get` gives the *cell* accursed by this cursor; `getPosition` gives the *media stream position* accursed by this cursor. `setPosition` sets the media stream position, and thereby implicitly sets the accursed cell as well: it is the cell that contains this media stream. `set` sets the accursed cell, and also makes the media stream position `null`, i.e., makes the cursor accurse no specific position inside the cell.

Cursor bias

Cursors in text documents are classically between characters, and have a bias, which is generally to the right. Let us consider the "f|oo" example again (the cursor is obviously between the f and the o), and assume the standard bias to the right-- we'll note "f[oo." The bias comes into play when there is an insertion at the cursor position: the cursor sticks with the character it is biased towards. Let's say we insert "bar" at the above cursor position; then we get "fbar[oo." Had the cursor been biased to the left, i.e. "f]oo," we would have gotten "f]baroo."

Now, a cursor cannot really be between two cells in ZigZag-- you cannot connect to a connection, you can only connect to a cell. Thus, we model the cursors by

accursing the character the cursor is biased towards: the cursor in "f[oo" is connected to the first o, while the cursor in "f]oo" is connected to the f. Now, we can say that a cursor accurses a *side of a cell*: "f[oo" accurses the left side of the first o; "fo]o" would accurse the right side of the first o. (XXX Image!) In addition to the cell in the media stream we accurse, we thus need to store the side we accurse. This is unique to cursors into media streams: when we normally accurse a cell, the "side" concept (poswards or negwards on the virtual media stream dimension) has no relevance.

Now, if we distinguish between cursors on a cell, and cursors into a media stream, we can say: Cursors on a cell are those cursors which do not have a side attached; cursors into a media stream are those cursors which do have side attached. Remember that a cursor accursing the cell containing the o would not be moved if the o were removed from the stream, but a cursor pointing into the media stream, e.g. the "f[oo" cursor, would be moved so that it is still inside the stream; e.g. if the first two characters were removed we would get "[o," where the cursor is on the single remaining o. What to do with a cursor can be determined by finding out whether it has an attached side.

However, we said above that a cursor pointing into a media stream also accurses the cell containing that media stream at the same time. So a cell pointing into a stream always also accurses a cell. On the other hand, a cursor can accurse a cell without pointing into a stream; that's the case when it has no side attached.

XXX explain better, make terms clearer! Also, anything else about cursors?

Markup

markup is connected to the stream on `d.markup-list`, and defined on `d.markup` (first cell on `d.markup` connects to the first marked-up character on `d.markup-list`, second cell on `d.markup` connects to the last marked-up character on `d.markup-list`)

markup stays with the marked-up ranges, even when text is cut out-- that is why the system dealing with spans needs to know about it

XXX

Optimization

a special dimension, which efficiently stores connections inside a span: in a-b-span.1-span.2-span.3-c (where span.x is "the xth element in the span), the connections inside the span are just stored as "span.1 to span.3 are connected." the other connections are stored regularly

Lookups

Given a space, it is possible to obtain a list of cells that overlap with a given list of spans. These can be used in various ways to display parts of texts that have links in different ways etc.

Later on, an enfiladic (tree-like) structure will be used to perform the cross-matching of the cell lists.

ZObs

It is going to be fairly common for a Java class to get its parameters from the structure. The ZOb mechanism is useful for easily creating such Java classes that read their parameters in a standard way. Basically, a ZOb is a Java class for which the instance variables are defined inside a `STRUCTPARAMS {}` block. There is some syntax support for specifying the number of elements required in arrays etc.

The point of the mechanism is to allow more latitude with the parameters later: for example, inserting them into the structure with descriptions, or caching the ZOb parameters read from the structure for larger ZOb systems to speed up the process.

Interactions

This section deals with the interactions between the various planned features. Unfortunately, this is one part that is not yet fully specced but at least we're speccing what the problems are. (XXX todo)

	Clones	Cursors	Versioning (access to old versions)	Slices	Content links	I18N
Clones						
Cursors						
Versioning (access to old versions)						
Slices						
Content links						
I18N						

Summary of special dimensions

d.clone

The clone dimension. All cells on a clone rank are clones of each other, meaning that their contents are enforced to be the same and changes to the contents of one are reflected to the others. For example, clones could be used to represent the same email address on a number of email headers, so that all those instances of the same email addresses would be linked to each other through `d.clone`.

d.cursor, `d.cursor-list` and `d.cursor-cargo`.

A set of pointing dimensions. This mechanism is used for instance by the views to point to the cursor which is at the center of the view. Moving the cursor is equivalent to inserting the maincell of the view poswards on `d.cursor` from the new position of the cursor (the accursed cell). `d.cursor-cargo` provides a mechanism for locking several cursors together.

File format

Note: This section describes the file format to be used in the new implementation of GZigZag, in src/. This file format is to be used from version 0.8.0 onwards. The file format is not yet finalized; the format currently documented in this section is not the final GZZ1 format and changes may well occur before release. Once this format is fixed, it will be mentioned here.

This section describes the GZigZag file formats that store the low-level structure. This is separate from the space description which describes e.g. the system list: the file format is lower-level still.

The file format is arranged as several layers, in order to make it more flexible in the future.

In order to retain some transparency for the user and a feeling of openness, the file is completely encoded in UTF-8 and all the control structures are ASCII characters. This way, if something goes wrong, it is possible to see what's in the files easily. Space efficiency is not really an issue since the stream can be gzipped if desired.

In order to make the format robust and easy to parse, again at the expense of brevity, all the nonterminals are self-delimiting and therefore the format can be parsed without any lookahead.

The current version number of this format is 1.

Cell identifiers

Identifier formats

Cell identifiers are currently defined as arbitrary octet sequences.

There are several ways to identify cells. In the format, a cell identifier is a single letter indicating the storage format, followed by the identifier.

We currently define only the following two ways to store cell identifiers:

'C'*(text-id)*' '
: The capital C is followed by a sequence of characters from the set [0-9a-zA-Z.:,+-]. This set may be expanded in future versions. A space (ASCII 32) character ends the sequence

'c'*(hex-coded binary string)*' '
: If the ID contains other data, it is represented as a hex-coded binary string.

The first format is intended for brevity and clarity; it is recommended that it be used if possible but it is possible to use only the second format.

It is likely that in the future there may be other formats to remove redundancy resulting from the space ids.

Examples. "Cabc012 " = the cell 'abc012'. "c616263303132 " = the same cell in the other encoding. Note the terminating spaces at the ends.

Semantics: local and global identifiers

Tjl Maybe some more explanation here of mediaserver etc?

The mediaserver id of a block to be written is not known when the block is written, since the id contains a cryptographic hash of the block's content. This poses an interesting problem for the file format: the ids of new cells, created in the current block, are of a different format than general ids.

The octet 0x2D ('-') is used as a separator between the block id and the local id. If the id starts with that character, it is local and equivalent to having the id of the mediaserver block in which the reference is made prepended.

Example. In mediaserver block XXX:

```
+
C-1_C-2_
```

following that, in mediaserver block YYY:

```
+
C-1_C-2_
CXXX-2_C-1
```

overall, the result is a chain of cells: "XXX-1" --- "XXX-2" --- "YYY-1" --- "YYY-2"

Simple dimension single deltas

The changes between two versions of a single simple dimension are described by a sequence of disconnection and connection operations. The basic format is **'-\n'*(disconnects)*'+\n'*(connects)*'0\n'** where the (disconnects) and (connects) are pairs of cells followed by a newline after each pair.

Each pair of cells refers to one connection either made or broken. It is forbidden for the same cell to appear twice in the same position (first/second) in the disconnect or connect list: the same cell can only be disconnected from one cell on each side and connected to one cell on each side.

The design for the delta is such that it is simple to reverse: simply exchanging the disconnects and connects produces the inverse delta.

Example. (underscore means space)

```
-
```

```
C1_C2_
C2_C3_
+
C2_C1_
C1_C3_
0
```

Assuming that the cells 1, 2 and 3 are at first arranged as 1-2-3, this delta rearranges them to 2-1-3 by deleting and creating two connections.

Cell content deltas

The changes to cell contents are not as easily reversable as the connection changes; therefore, to allow better compression, there are two different variants of the content delta format: the irreversible and reversible formats.

The records here are either 'k' followed by a cell id and two content specifications: the contents before and after, and 'K', followed by a cell id and the content after. The same cell may occur in a single delta only once.

The content specifications are a somewhat difficult at the moment since it has not been exactly specified what a cell can contain. Because of this, we shall define two tentative content types: UTF-8 strings and spans.

The UTF-8 strings are escaped by transforming all backslashes into double backslashes and all instances of the newline character (0x0a) into a backslash and the character 'n'.

The spans are stored as a sequence of hex digits (the mediaserver ID), a colon and a comma-separated list of numbers to be interpreted by the span type, and finally a newline.

Space inclusions (primitive slices)

Space inclusion is described below. When including a space, it is necessary to prefix all references to that space by something, since the included space could be or contain e.g. an older version of the current space, or two included spaces could contain some common space etc.

One important point is that all included spaces are associated with a particular cell in the current space. That cell is then used as a prefix in the cell id, with the format

```
(cell):(cell)
```

where the first cell is the prefix cell and the second is the cell inside that included space.

Example. First, include the space A using ZZ structure (currently interfaced by `SimpleCompoundSpace.include()`). Then, connect cell 42 in that space to the cell 7 in the current space (of course, this is in a different section of the file). Note the use of the : cell format

```
+C1:42_C7
```

Note: for dimensions, the global identifier with no prefixes is always used.

Transcopy deltas/sections

Benja: XXX update!

Benja: The following section is a proposed extension to the described-- and already implemented-- file format. It assumes some background information available in `Documentation/misc/cellids` (the content there should probably be moved here or to another .wml document). I consider transcopies to be vividly important and want to implement them... now, so please comment.

 I first read that wrong: I thought that you had implemented this design already ;)

A transcopy section in the diff is not really a delta. It contains a space version to transcopied cells from, and a list of pairs the first element of which is the cell ID of the to-be-transcopied cell in the space version transcopied from, and the second part of which is (in the language defined in `Documentation/misc/cellids`) the tlid, the local part of the transcopy ID, appended to the cell's cid, or constant ID, in the new space. (The tsid, the space version part of the tid, is implicitly given through the ID of the block the cell was transcopied in.)

 First: the ids should be discussed in either this document or DesignProblems. Second, I'd really like to see this matched with the use cases that Tuukka is working on.

All transcopy sections in a single delta should come before all content and all dimension sections. This is because the transcopied cells' content and connections cannot be changed before they are transcopied, but can well be after.

I propose the following simple format: A transcopy section is a number of lines each containg one of the pairs as described above, ended with a line that contains only '0'. Each line contains two cell IDs; however, the second ID is not really a cell ID but a local ID part, namely the tlid as described above.

 When proposing syntax, it's be really nice to have it in the same EBNF as there is below.

A single delta on multiple dimensions and content

The change in a whole space between two versions is encoded in a single delta on multiple dimensions. **Note: for identifying dimensions, the global identifier with no prefixes is always used.**

Transclusions of spans

The way spans work are described in another section above. The connections on the VStream dimension are saved as ordinary dimension diffs. However, the way

the spans are *transcluded* into a space needs some special handling, because we do not want to specify each single cell to be transcluded, but rather the *range* of transcluded units.

A section of span transclusions has the following general format:

```
spans           : ('s' cell-id block-id ' 'number ' ' number '\n')* '0\n'
                ;
single-delta-part:
                .
                .
                .
                | 'S\n' spans
                ;
```

where the `cell-id` is the id of the transclusion, `block-id` is the scroll block we transclude from, and the numbers are the indices of the first and last units we transclude. For example, `s00918B 14 17` would transclude the cells `00918$14`, `00918$15`, `00918$16`, and `00918$17` (the units 14-17 in the scroll block).

Collected syntax

```
/*
 *      Basics
 */
number: [0-9]*.[0-9]* ;
hexdigit: [0-9a-zA-Z];
id-char: [0-9a-zA-Z:,+-]
cell-id : 'C' id-char* ' '
        | 'c' (hexdigit hexdigit)* ' '
        ;
block-id: (hexdigit hexdigit)*
local-cell-id   : cell-id ;
escaped-utf-string      : [^\n] * ('\\' [n\\] [^\n]*) '\n' ;

/*
 *      Simple dimensions
 */
cellpair: cell-id cell-id '\n'
                ;
simple-dim-delta: ('-' cellpair)* ('+' cellpair)* '0\n'
                ;

/*
 *      Content
 */
content-spec    : 't' escaped-utf-string
                ;
content-change  : 'k' cell-id content-spec content-spec
                | 'K' cell-id content-spec
                ;
content-delta   : (content-change '\n' ) * '0\n'
                ;

/*
 *      New cells
 */
new-cells       : ('n' local-cellid '\n')* '0\n'
                ;

/*
```

```
 *      Transcopies
 */
transcopies     : ('t' cell-id '\n')* '0\n'
                ;

/*
 *      Transclusions from spans
 */
spans           : ('s' cell-id block-id ' 'number ' ' number '\n')* '0\n'
                ;

/*
 *      High-level structure
 */
single-delta-part: 'D' cell-id '\n' simple-dim-delta
                | 'E\n' content-delta
                | 'N\n' new-cells
                | 'T' cell-id block-id '\n' transcopies
                | 'S\n' spans
                ;
single-delta    : single-delta-part* '00\n' ;

gzzfile         : 'GZZ1\n'
                  number '\n'
                  block-id '\n'         /* Id of previous space */
                  single-delta ;
```

Cell identity

In the new "src/" implementation, identifying primitives is based on cell identity instead of the contained string. This, with the immutable mediaserver blocks brings out a whole slew of issues we have to solve with regards to cell identities between different versions, connecting different spaces together etc.

Some problems to solve

In this section, we'll attempt to discuss some of the problems that need solutions.

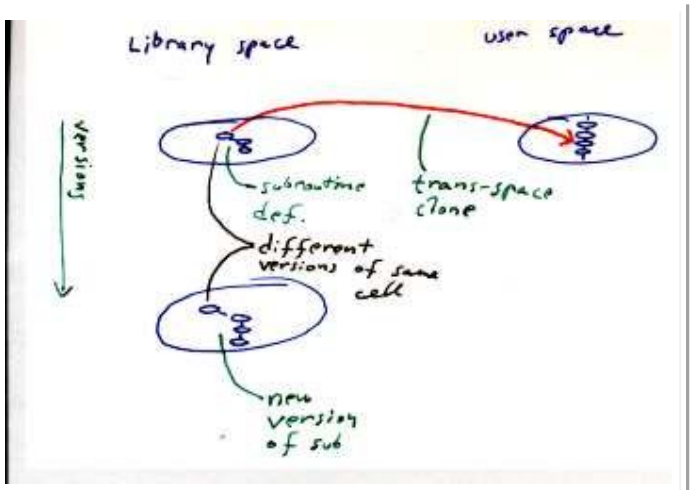
Primitives

We wish to define dimensions and types, such as "Lambda expression", in a space and use them in other spaces.

Tjl I'd prefer the term "Primitives", as it's not only dims and types but procedures, operators etc.

Versioning Clang programs / primitive sets

When we produce new versions of the definition space, old code and new code should be compatible at least in areas of no change.



The program versioning problem. A cell from the library is cloned across space to the user space.

Tjl This passes by very briefly a really major set of problems. What determines what version of a procedure needs to be called in a different space, if we only use the original IDs? For example, in the figure, when the library is modified, what happens? Where is it decided what version of the original cell the clone in the user space corresponds to? Here, we need to provide all features a good shared library system has: simultaneous versions, testing whether an upgrade works, upgrading the version of the library etc.

R^2 Original ID's? Why not use a cell, which specifies whose clone it wishes to be, and the `d.clone` connection is generated by a space part? The specification could be e.g. "the latest version of procedure X in the library that has features a, b and c that are compatible with those from version xyzzy".

Tjl "a cell which specifies whose clone it wishes to be"!?!?!?!?!?!?! That has a LOT of problems; clones should not change from what they have been cloned at all. The version, I think should be the version of the included space which may be upgraded... There, you might have something like this, autoupgrading etc. but for single cells that sounds horrible.

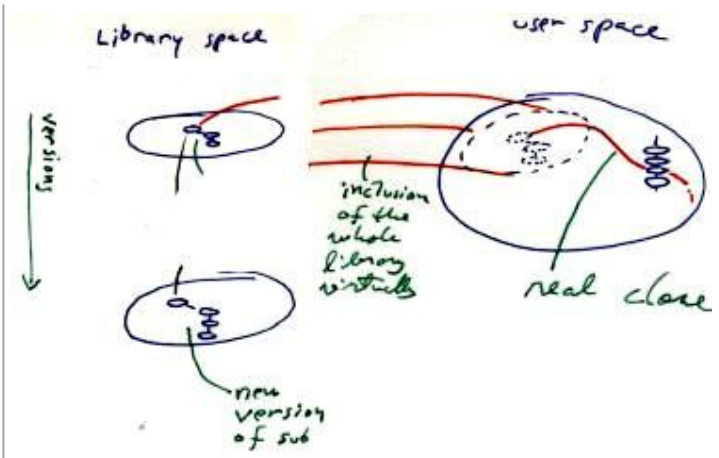
Modifiability

Others should independently be able to add their own modifications to the definitions, without interfering with our work. This would be things like adding a new expression type into a simple imperative Clang, or a new view type to a list of view types, etc. [XXX missing stuff here]

Tjl A **lot** of missing stuff ;) I don't understand this at all

Space composition

Given multiple spaces (e.g. from different slices, or autogenerated), we wish to combine them into a new space in some user-defined way. The user must be able to define this in the structure. Examples of this would be the preflets and autogenerated document structure tree in Nile.

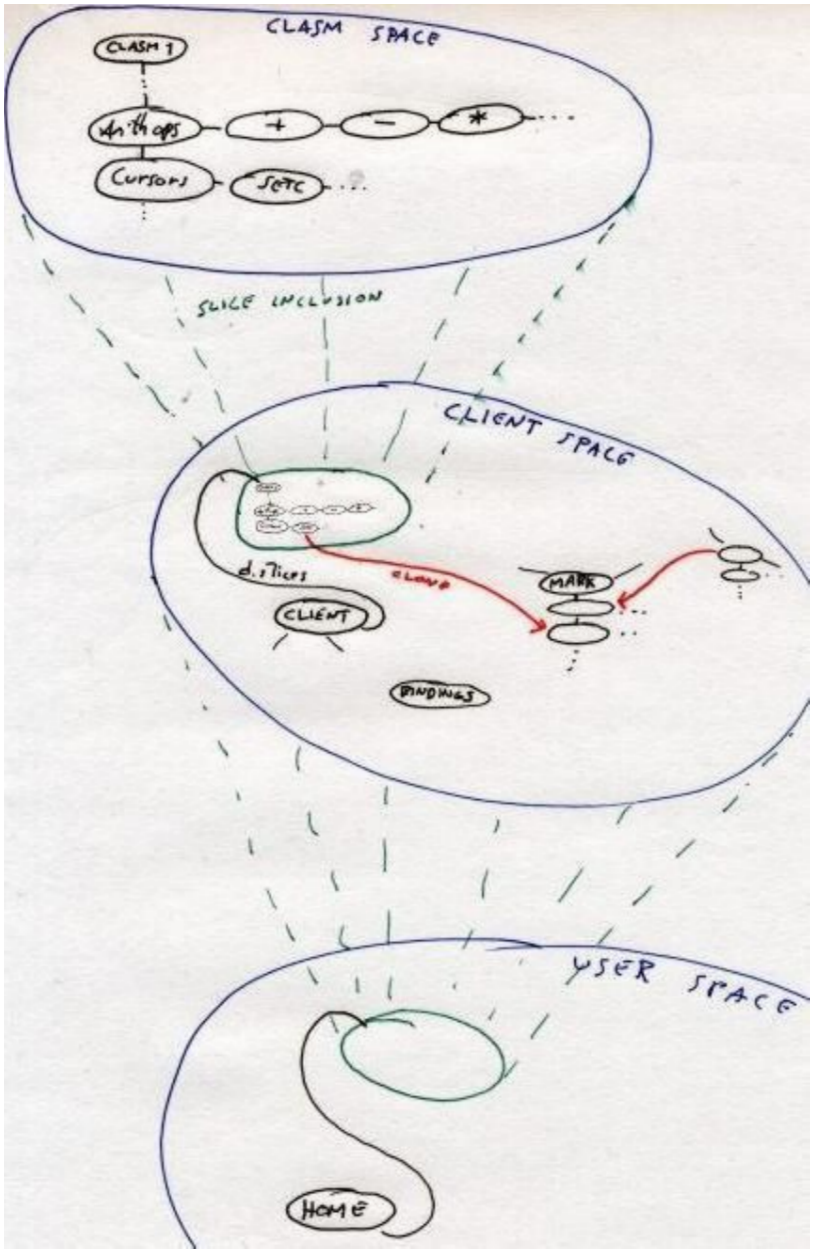


How the above versioning and this inclusion are related. The set of problems is different if the library is included by transcluding the whole space.

Tjl The real problem you're trying to bring out here is that the user needs a way to refer to another space inside the one space? This is not only a structural problem but also a presentation problem: any method can be chosen as long as the user presentation is suitable. See figure.

Slices, transcopying, inclusion

In this section, we specify the simplest slice model which shall hopefully remain upwards compatible with later, more refined functionality but should be powerful enough to support the first clients. The overall idea is shown in this figure.

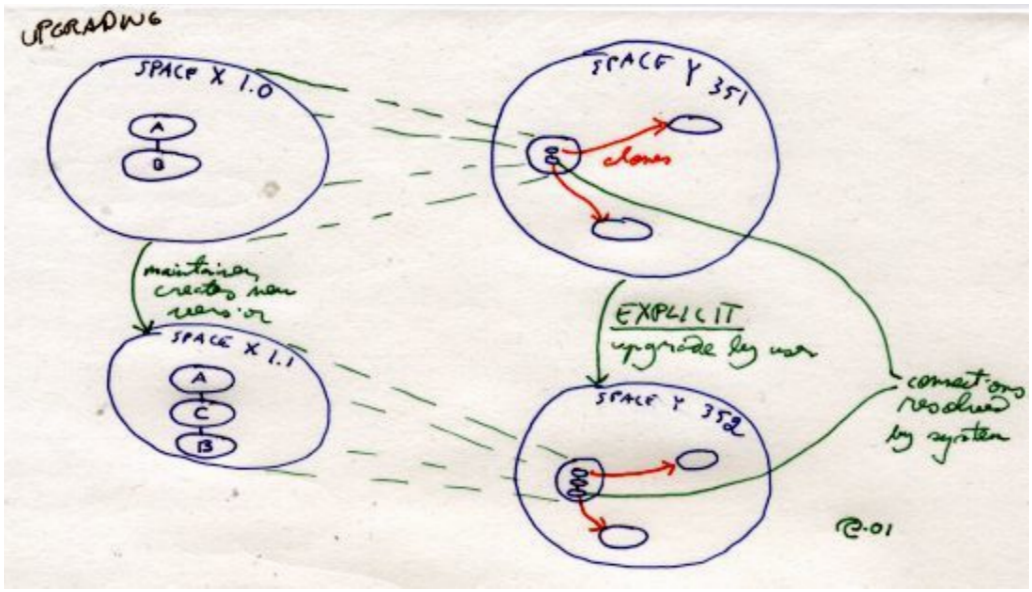


The analogy to shared libraries and dynamic linking must be stressed.

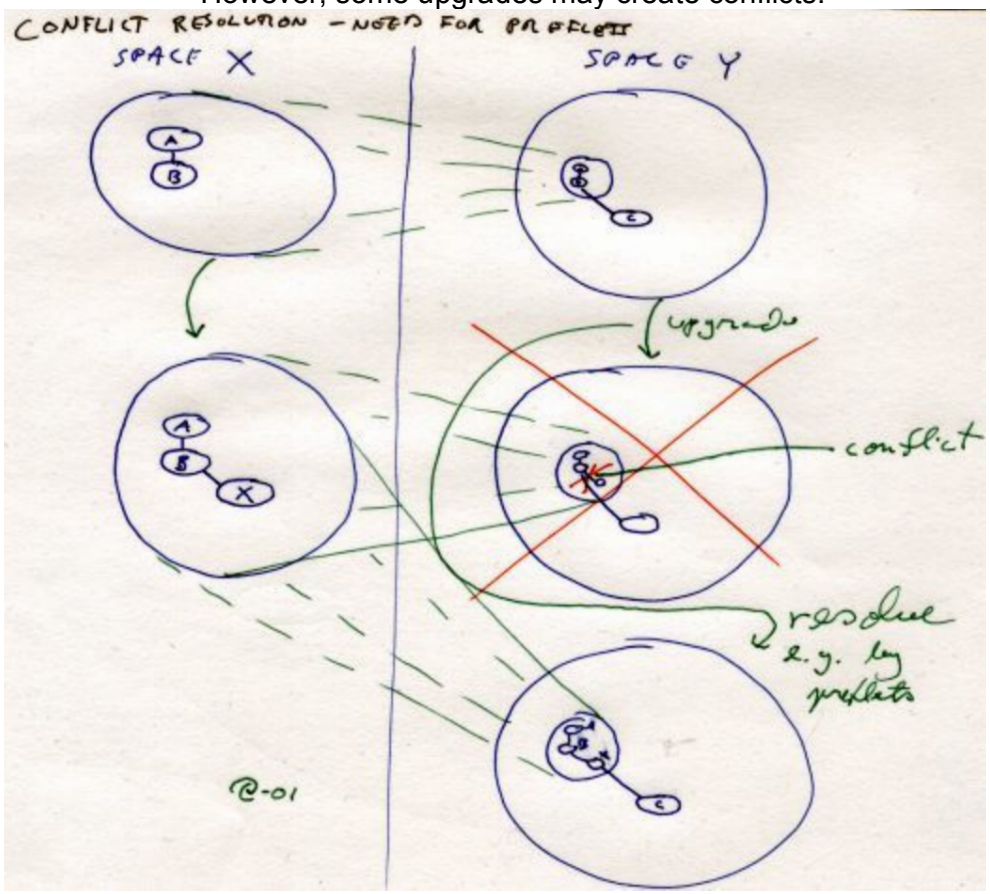
The lowest-level (hard-coded!) space defines the cells that represent the CLASM primitives in the ZZ space. The Client Space, which contains basic subroutines for user interaction and basic sets of bindings, includes the CLASM space in order to use the primitives in subroutines by cloning them. This space is then included in the user space, which contains the user's data.

Versioning, changing, preflets

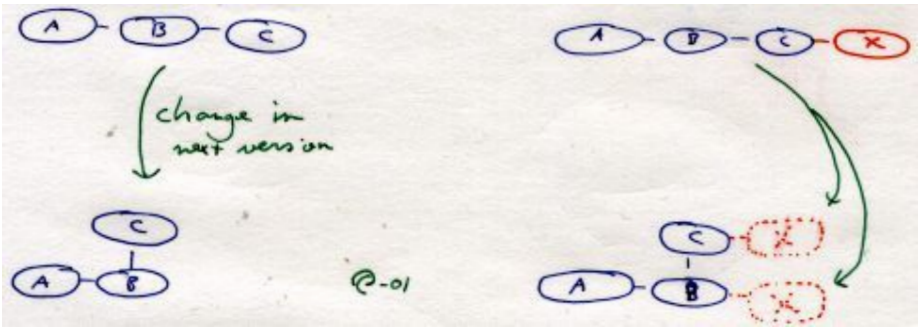
However, including slices brings on complexities. First of all, we do want to be able to upgrade the included slice:



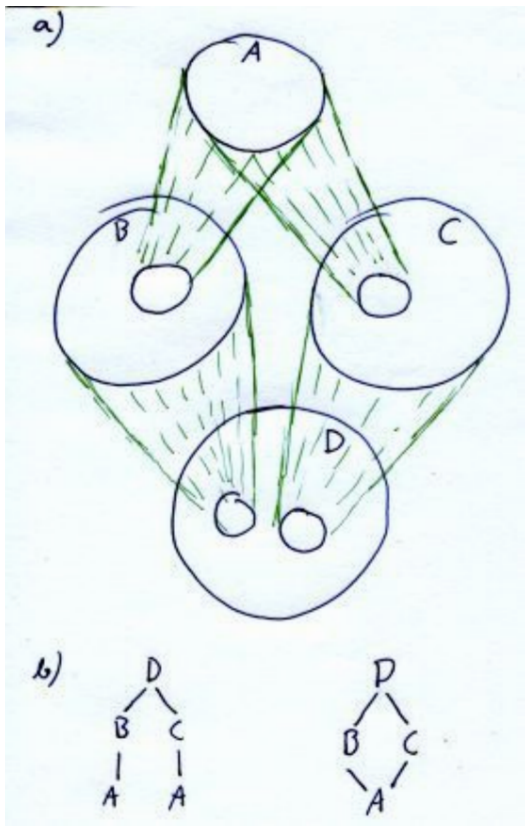
However, some upgrades may create conflicts:



Preflets are Ted's mechanism for expressing the connections in such ways that conflicts can be resolved. However, there are several types of preflets and simply connecting cells does not give enough information about the intended connection:



Containment

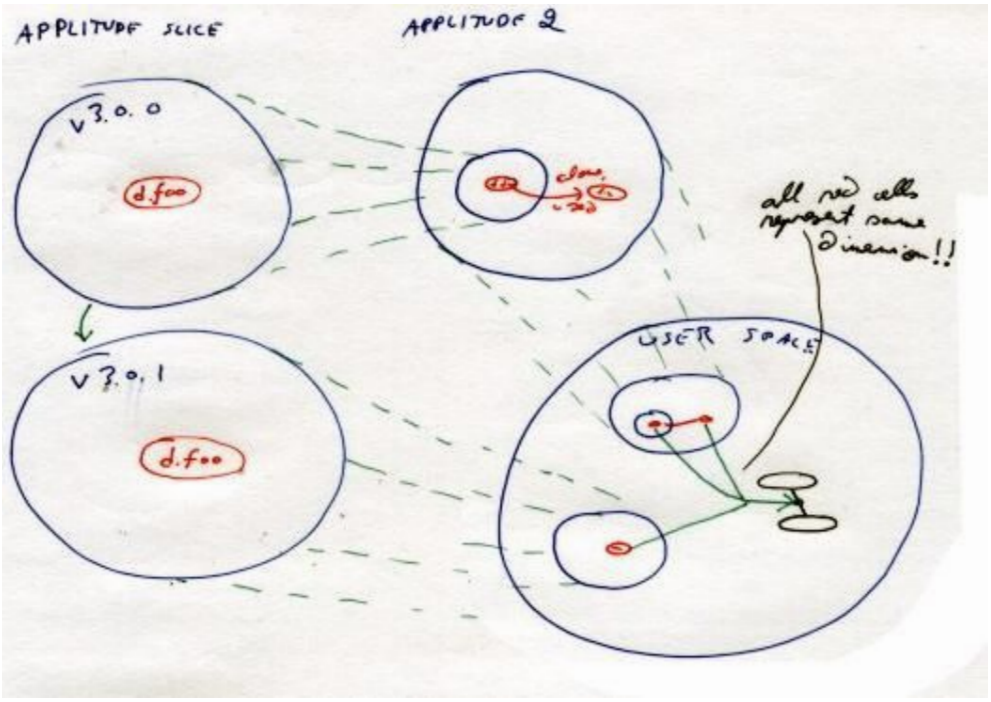


a) The space A is included into space D as a part of two different spaces. b) Two possible models: either keeping the different copies separate or unifying them.

One related question is whether spaces are shared or unified when they are included in different included spaces. For example, here the space A is included in both B and C. Space D includes both B and C. Now, are there two different instances of A inside D or a single, shared instance.

In the first version, we would like to use the former model: the spaces are always included verbatim, with no consideration of lower-level subspaces included.

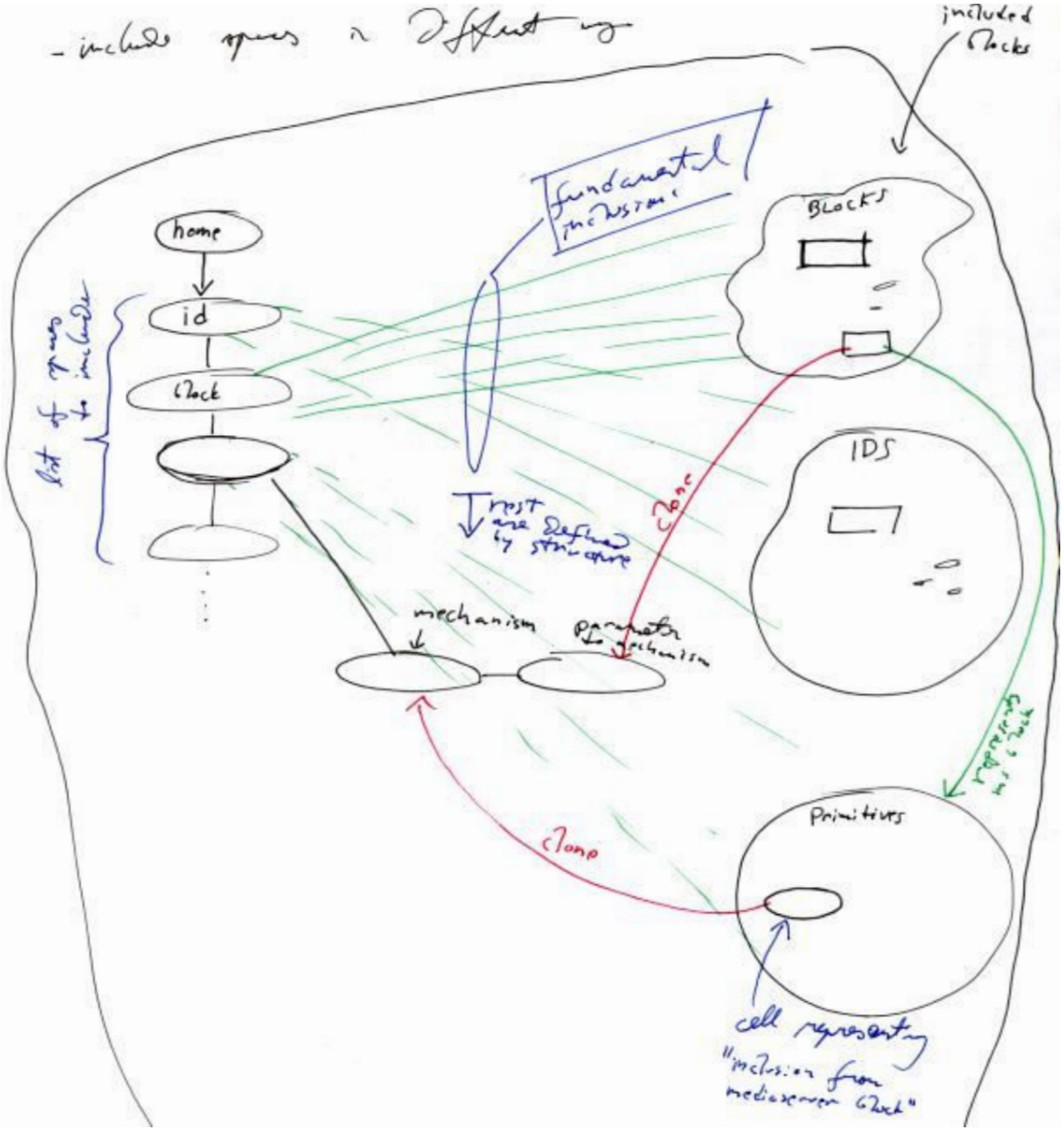
However, there are some interesting problems here, related to dimensions. Consider the image

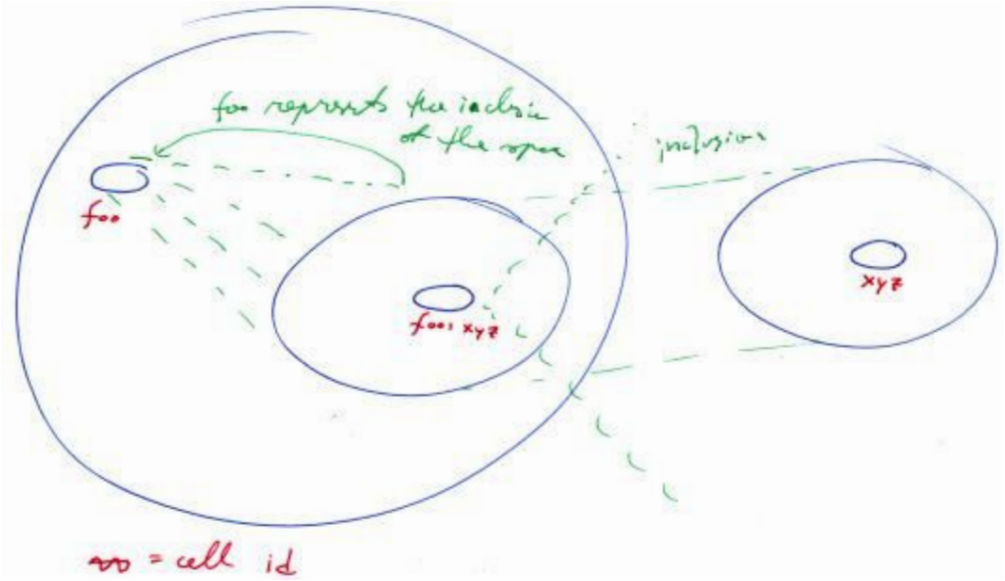


Here, it is easily seen that in the user space, all the red cells should represent the same dimension. For clones, that is natural, but for the cells that come from different versions of the space via different routes, this is less obvious.

This consideration also explains where `d.clone` comes from; we have tacitly assumed its existence and globality in the above but not discussed e.g. where the cell which contains it comes from. With the above considerations, it is easy to create a basic space included in all spaces which contains the cell representing the clone dimension and possibly some other such global dimensions.

Implementing inclusion


- includes spaces in Defining
included Blocks
fundamental inclusions
home
id
Block
list of spaces to include
Blocks
IDS
must are defined by structure
clone
mechanism
parameter mechanism
Primitives
clone
cell representing "inclusion from mediaspace Block"


foo represents the inclusion of the space
inclusions
foo
foo: xyz
xyz
= cell id

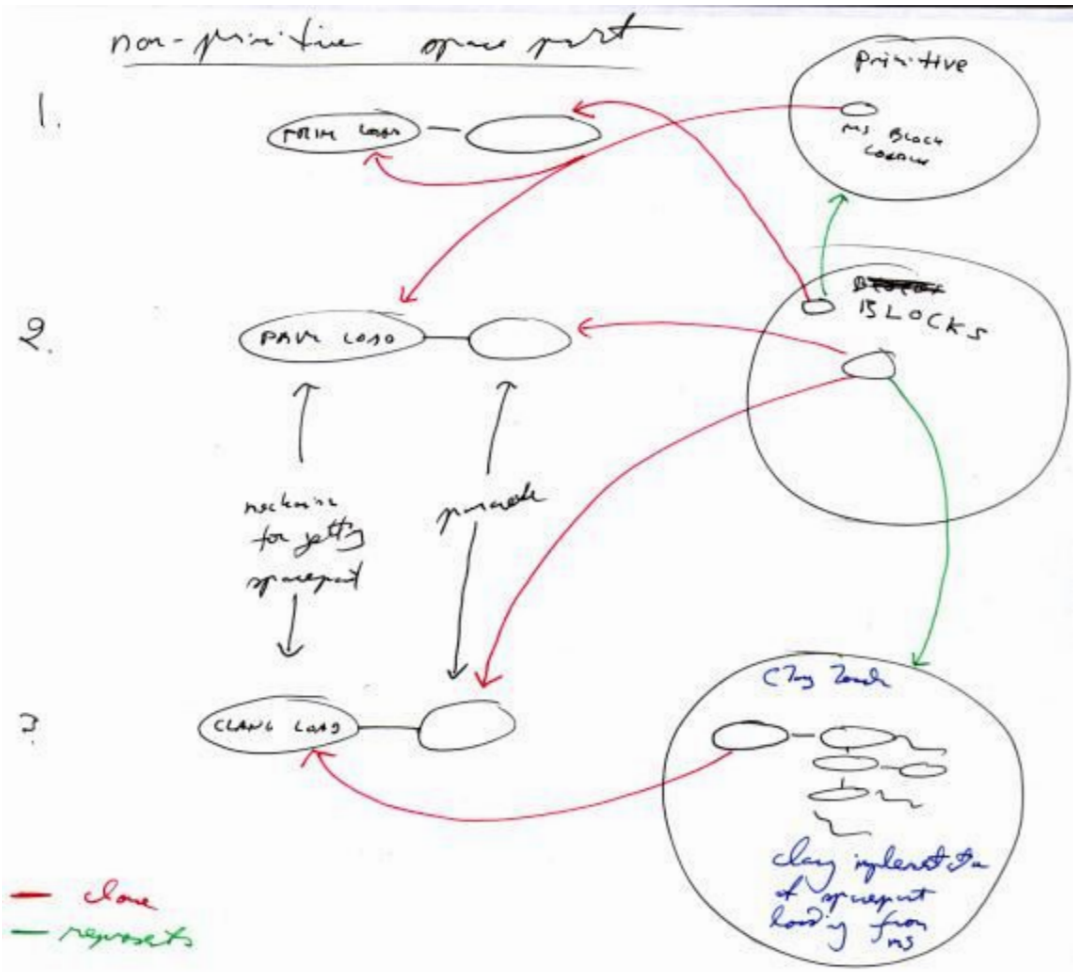


Image spans

We really want to do image spans containing parts of postscript/PDF files. The problem is that creating the images from these files is time-consuming but caching the images is space-consuming. We need to be able to do tradeoffs here.

For example, when paging through a multi-page document, usually ps/pdf viewers show the pages fairly slowly. We want INSTANT response, but the initial image is allowed to be of lesser resolution. This way, "leafing through" the document is easier: you always see what page you are on, unlike with normal ps viewers.