CLIM 3
A new interface manager for Common Lisp

Robert Strandh

July 30, 2012

Chapter 1

Preliminaries

This document is work in progress. It contains some ideas about the successor of CLIM 2, the Common Lisp Interface Manager. CLIM 2 is in many respects a very good specification. We believe, however, that the reason for many design decisions were related to limitations of the performance of computers at the time the specification was written; limitations that are much less severe these days.

We have attempted to simplify many aspects of the CLIM 2 specification, often according to the *issues* that are listed in it. One major such simplification is that we have merged three concepts: *sheet*, *output-record* and *design* (to some degree).

Chapter 2

Changes from CLIM 2

2.1 General

This section explains the general changes that we consider for CLIM 3 compared to CLIM 2. Following sections will contain specific changes with respect to particular features of CLIM 2.

We intend to act upon one of the "major issues" indicated in the CLIM 2 specification, namely a suggested merge of the *sheet* and *output record* concepts. This will greatly simplify the design of CLIM 3.

There are many issues with the CLIM 2 specification. For instance, what happens to an output record (which is rectangular according to the specification) when it is in a pane that has an arbitrary sheet transformation that does not preserve this shape? For CLIM 3 we want to clarify what happens in such situations, and eliminate it as a part of the specification if the concept is hard or impossible to implement.

Several aspects of the CLIM 2 specification make it hard to localize applications. One such aspect is the fact that command-line names are part of the command table, requiring a different command table for each language. We want to clean up such aspects so as to prepare CLIM 3 for the possibility of writing applications using different languages.

By clarifying the semantics of *zones* (which replace CLIM 2 *output records*), especially with respect to the exact time when the effect of manipulating the hierarchy of these objects is visible, we avoid the almost inevitably quadratic algorithms involved in manipulating CLIM 2 output records.

We keep the layout protocol, but clarify when it is invoked. We replace space requirements by vertical and horizontal elasticity functions.

We eliminate mirrored sheets as an explicit abstraction of CLIM 3 and let each backend decide on a case-by-case basis whether a zones should have a mirror or not.

We eliminate alignment and spacing options to the layout panes and instead provide a zone type that is very rigid (for spacing) and a zone type that is very elastic (for alignment).

2.2 Regions and Transformations

In CLIM 2 this is a very general concept. We will probably keep these, but the CLIM 3 equivalent of a CLIM 2 sheet (called a *zone*) will not admit arbitrary regions and transformations.

2.3 Sheets

2.4 Commands and command tables

2.5 Internationalization

Define a classes for country, language, printing of numbers, units, etc, and locale classes that inherit from subclasses of those.

2.6 Text styles

We keep the concept of text styles, but we may remove the concept of merging text styles. We add the possibility of naming font families. We will specify that text-style-ascent, text-style-descent, text-style-height, and text-style-width may only be called as part of the layout protocol, because they are only defined when the zone is attached to a port.

2.7 Layout protocol

We guarantee that the layout protocol will only be run when the zone hierarchy is attached to a port. Thus, when a text zone is asked to provide space requirements, then it is safe for it to ask the size of the text, because the text style then has a mapping to a device font.

Chapter 3

Introduction

All terminology is preliminary.

At the lowest level of abstraction, CLIM 3 uses a *canvas* which is an axis-aligned rectangular area of opaque *pixels* supplied by the underlying windowing system. Drawing on this canvas means taking some arbitrary subset of pixels on it, and modifying their color values in some way. The result thus depends not only on what was drawn, but also what was there before. The *order* in which drawings are made is thus important. The canvas uses integer coordinates.

Applications do not draw directly to the canvas, and in fact CLIM 3 does not have any representation of the canvas. Instead, CLIM 3 manages a *hierarchy* of *zones*. A zone is an axis-aligned rectangular area. Each zone manages its own abstract *drawing plane* which is a potentially infinite plane with its origin in the upper-left corner of the zone. The position and size of a zone are defined with integer coordinates, and the position is relative to its parent zone. The effect of drawing a zone is *clipped* by its parent zone, i.e., a request to draw a zone is accompanied by a clipping region, and zones are not allowed to draw outside this region, because if they did, the result would be incorrect.

When a hierarchy is attached to a *port* (representing a display server), the *backend* takes this hierarchy of zones and *realizes* each zone in the hierarchy in some device-specific way. Some backends may choose to create a window for each zone, and some others may chose to create a single window for the top zone.

Zones are either *atomic* or *compound*. An atomic zone represents an elementary graphics object such as a line, a rectangle, an ellipse, or an image. A compound zone contains a *sequence* (not necessarily in the sense of Common Lisp) of *children*. The children are other zones and the sequence represents the *order* in which the children are drawn.

In terms of zone composition operations, zones on the same level of the hierarchy are composed using the *compose-over* operation, in the order determined by the sequence of zones, and for a compound zone, the result of the composition of the children is composed with the parent, which is considered to be a zone with opacity 0 outside the zone and 1 inside it. An *opaque compound* *zone* is computed as if its bottom child were a totally opaque zone with some *background color*.

Most zones are not created with exact sizes, because the exact size of a zone may depend on the backend. This is in particular the case for a *text zone* the size of which depends on the *font* used to render it, and the font is specific to each backend. Instead, when requested by the parent zone, each zone must generate a *space requirement* object. As a function of this space requirement, space is then allocated, and the zone must adjust itself to that space. Creating space requirements and allocating space is part of the *layout protocol* and the layout protocol is invoked whenever a zone hierarchy is attached to a port and then after each event that might potentially result in a modification to the zone hierarchy, or when the application explicitly requests it because the zone hierarchy might have changed as a result of some application activity. In most cases, CLIM 3 will detect that there are no or very few modifications to the zone hierarchy, and minimize the actions that need to be taken as a result.

Applications can manipulate the zone hierarchy in several different ways:

- They may explicitly insert or delete zones. In this case CLIM 3 may try to minimize the areas of the canvas that need to be redrawn by re-using pixel colors that are known to be the same as last time around the command loop. A special case occurs when the application inserts new zones *at the top of all existing zones* each time around the command loop, because then the existing pixels do not have to be redrawn.

- Simpler application might generate new zone hierarchies every time around the command loop. In this case, CLIM 3 will have to redraw the entire contents of the zone. To minimize flickering, applications may request that *double buffering* be used in this case.

- An application may produce a new sequence of children for each iteration of the command loop such that some or all of the new zones are *the same* as the existing ones. CLIM 3 will compare the new sequence to the old one, and again minimize redrawing.

- An application may *move* a zone. Since the canvas uses integer coordinates, CLIM 3 might then be able to move the pixels without requiring the application to generate new output. This is a particularly important optimization when *scrolling* is required.

Chapter 4

Optimal redrawing

4.1 Introduction

The purpose of optimal redrawing is not mainly to touch as few pixels as possible in the visible area of the screen, even though it can do that as well. Instead, the purpose is to simplify applications by having CLIM 3 manage a very large number of zones, most of which are invisible at any point in time. Imagine for instance a word processor or an editor for music scores. Such an application might produce hundreds of pages of output. Instead of leaving the burden to the application writer to determine what part is visible, and how large the scroll bar should be, it would be good if the application can produce the entire output, and let CLIM 3 deal with scrolling and incremental modifications to the output.

Such an application might be written to produce a hierarchy of zones. At a high level, each page might be a zone. Each such page can contain paragraphs or, in the case of a score editor, *systems*, then lines, words, and perhaps individual characters. This output is visible through a small viewport that might cover a single page, or at most a few pages. The main purpose of the algorithm presented here is to avoid invoking code to draw parts of the output that are currently invisible and parts that have not changed since the previous time. For instance, after a single modification of a page of text, most paragraphs might be intact. The algorithm described here is able to avoid redrawing such paragraphs.

In other words, the application writer does not explicitly manage visibility, and can write the application as if every part of the output is visible.

The algorithm makes two passes over a hierarchy of zones, the first one from top to bottom, and the second one from bottom to top. For this reason the sequence of zones is probably best represented as a doubly-linked list.

In the first pass, a region to be redrawn is computed. This region starts out as empty and is potentially added to for each zone in a sequence.

In the following algorithm, the region is clipped by zones that are not visible after the redraw. A zone can be in one of six states:

- Unchanged. This means that the zone has already been drawn in the past, and that it hasn't been moved or altered since the last time it was drawn.

- Deleted. This means that the zone has already been drawn in the past, but is no longer supposed to be. A deleted zone might also be marked as moved and/or modified, but "Deleted" takes precedence. The region that is needed by the redraw algorithm is the region the zone had last time it was drawn, i.e., before it was moved or modified.

- Inserted. This means that the zone has not been drawn in the past, but is supposed to be now. An inserted zone might also be marked as moved and/or modified, but "Inserted" takes precedence. The region that is needed by the redraw algorithm is the region that the zone has after having been moved or modified.

- Moved and not modified. This means that the zone has already been drawn in the past, but that it has simply changed positions since the last time it was drawn.

- Modified but not moved. Only a compound zone can be marked as modified, indicating that a child zone has been inserted, deleted, moved, or modified.

- Moved and modified. In this case the zone is treated as deleted from its position when last drawn, and inserted at its final position.

Applications are responsible for:

- Marking a zone as "Modified" when any of its children is either inserted, deleted, moved, or modified.

- Marking a zone as "Moved", whenever it is moved.

The CLIM redraw algorithm provides the following services, so the application need not do so:

- Marking a newly created zone as "Inserted".

- Automatically detecting that a zone has been deleted, and marking it as "Deleted".

Only newly created zones can be inserted into a higher-level zone. In particular, applications are not allowed to change the stacking order of zones.

In the first pass, the redraw algorithm computes the region to be redrawn. This is done from the top zone to the bottom zone. In each iteration, it modifies the region to be drawn and passes it on to the next iteration, according to the following cases:

- If a zone is unchanged, then pass on the region without any modification.

- If a zone has been deleted, then add its area (i.e, the area it occupied when it was last drawn) to the region.
- If a zone is new (has been inserted since the last redraw), then if it is covered by some other visible zone, then add the area that is covered to the region. This is an important optimization, because when a new zone is drawn on top, what is underneath does not need to be redrawn.
- If a zone has been moved (whether also modified or not), then in the most general case, treat it as one deleted and one inserted zone.
- If a zone has been modified but not moved, recursively traverse the children.

In the second pass, the region that has been computed in the first pass is used to draw the zones from bottom to top.

- If a zone does not overlap the region, then do nothing
- If it does overlap the region, and it is an atomic zone, request an *redisplay* of the relevant region from the application.
- If it does overlap the region and it is a compound zone, recursively traverse its children.

4.2 Details of algorithm

The algorithm for optimal redrawing is divided into two parts. The first part computes the region to be redrawn, and the second part redraws that region.

To compute the region to redraw, the algorithm manages three regions:

- A region called `region-to-redraw`, which is the region that ultimately becomes the final result of the algorithm. This region grows as the algorithm traverses zones from top to bottom. The initial value for this region is `+nowhere+`.
- A region called `visible-region-after` which is the region that is visible after the redraw. This region starts off as `+everywhere+` and may decrease in size as a result of opaque zones.
- A region called `visible-translucent-covering-region-after`, which is a region containing potentially translucent zones *after* redraw. This region allows for an important optimization in that when a region needs to be redrawn and it is not covered by any translucent zones, it can be drawn on top of what is already there. If, however, it has translucent zones on top, the region needs to be "erases", meaning all zones in that region need to be redrawn from bottom to top.

The algorithm to compute the region to redraw thus takes four arguments: a zone and the three regions indicated above. It traverses the zones from top to bottom, and in each step, it modifies the three regions, and passes on the modified regions to the next iteration. A complete description of the algorithm must specify for each type of zone how the regions are modified.

For an *opaque* zone, the regions are modified as follows:

- `region-to-redraw`. If the zone has been deleted, then add its visible subregion to `region-to-redraw`. Otherwise, this region stays the same.
- `visible-region-after`. Of the zone has been inserted, or is the same, from this region we subtract the region of the zone, because of the opacity of the zone.
- `visible-translucent-covering-region-after`. This region is treated the same way as the previous one.

For a *atomic translucent zone*, the regions are modified as follows:

- `region-to-redraw`. This is an atomic zone, so it is either inserted, deleted, or unchanged. If it is unchanged, this region is not modified. If it is deleted, the visible part of the region of the zone is added to this region. If it is inserted, the visible part of the region of the zone that is also covered by translucent zones. The part that is not covered by translucent zones can be drawn on top of existing pixels without the need to redraw. Thus, we first compute the intersection of the inserted region of the zone with `visible-translucent-covering-region-after`.
- `visible-region-after`. This region does not change for translucent zones.
- `visible-translucent-covering-region-after`. If the zone has been inserted or is unchanged, then the visible part of the zone is added to this region. This region does not change for a deleted zone.

Finally, the complicated case is for a *compound translucent zone*. We would like to avoid scanning the children if possible, at the risk of having more zones redrawn at the end. Individual zones will indicate that a minor change has been made to its children by marking it modified and that a major change has been made by replacing the old zone by a new one, in effect deleting the old one and inserting a new one. We consider a compound zone to be relatively densely covered by translucent children, so that the entire region of the zone can be passed on as being a translucent covering region.

- `region-to-redraw`. If the zone is unchanged, we pass this region on unchanged. If the zone has been inserted or deleted, we treat this region in the same way as for an atomic translucent zone. Finally, if zone has been modified, we recursively consider the children of this zone. The initial value of this region for the children is the region of the zone after redraw intersected with the union of the region of the zone before and after redraw. The result is added to `region-to-redraw` and passed on.

- `visible-region-after`. This region is always passed on unchanged as with translucent atomic zones.
- `visible-translucent-covering-region-after`. Unless the zone has been deleted, the visible part of the zone is added to this region.

Chapter 5

Output recording

5.1 Introduction

Applications typically maintain some kind of data structure that is modified as a result of executing a CLIM 3 *command object*. To generate visible output from such a data structure, the application will have to produce a hierarchy of zones that CLIM 3 can draw on the canvas. CLIM 3 facilitates this task by letting the application programmer use seemingly low-level primitives such as `draw-line` and `draw-text` and automatically converting them to zones. Also, CLIM 3 maintains a *cursor* indicating the position of output within the parent zone, making it possible for the application to use ordinary Common Lisp output primitives such as `print` or `format`. Each line of text is automatically converted to a zone.

A simple application such as an IRC client or some other application that generates an increasing amount of output at each iteration of the command loop, might indicate to CLIM 3 that the hierarchy of zones of some pane should not be cleared each time around the command loop, and that each piece of output should simply be added to the hierarchy. Such an application might occasionally explicitly request that the hierarchy be cleared, but CLIM 3 will never do it automatically.

5.2 Incremental redisplay

Applications can be written with various degrees of performance in mind. A very basic application may simply clear the application pane and traverse the entire application data structure at the end of each iteration of the command loop and generate fresh output for its entire data structure. For instance, a text-editor application might have a data structure consisting of paragraphs, lines, words, and characters. Such an application might scan the entire data structure and generate a hierarchy of zones corresponding to its internal data structure.

A simple application such as the one above might be improved in terms of performance by taking advantage of the possibility of CLIM 3 to compare two hierarchies of zones, one at the beginning of an iteration of the command loop, and the other one at the end. By a slight modification to the application logic, one might insert a test for each zone to be generated that will verify if the contents of the zone is unchanged since the previous iteration of the command loop. If this is the case, the entire generation of the output can be omitted and CLIM 3 can reuse the existing zone instead. The advantage of this technique is that it requires only minor modifications of the very simple application logic that consists of scanning the entire data structure each time around the command loop, and that performance can be quite acceptable, even for fairly large data structures. Imagine again the text-editor application above. In most cases, a single line in a single paragraph has changed. The zones generated by most paragraphs can thus be reused. While each paragraph has to be tested, most tests will result in no further traversal of the paragraph being necessary. The time spent by the application is thus linear in the number of top-level data-structure elements (paragraphs) in most cases.

A more compound application will manipulate the hierarchy of zones explicitly. For instance, a word-processor application might have a data structure consisting of chapters, sections, paragraphs, words, and characters. A complicated incremental page-layout algorithm might convert this data structure to another one, consisting of pages, paragraphs, lines, and words. For performance reasons, the page-layout algorithm needs to be incremental, so the application needs to maintain a copy of the resulting data structure and only update it when necessary. An application of this degree of complexity would simply map each element of the resulting data structure (pages, paragraphs, lines, words) to a CLIM 3 zone. At each iteration of the command loop, such an application can still take care of the capability of CLIM 3 to compare two data structures, by generating a new hierarchy each time around the command loop. Most of the top-level zones (corresponding to pages) will be the same, and within the pages that have changed, most paragraphs will be the same, etc. CLIM 3 is thus capable of minimizing the update of the screen at a cost that most of the time corresponds to iterating over each top-level zone.

Chapter 6

Command loop

CLIM 3 maintains a *command loop* as follows:

- Acquire a *command object*.
- Acquire the arguments of the command object.
- Execute the command object on the arguments.
- Call `redisplay` on the top zone. The result of calling `redisplay` might be a new zone or the same zone, modified or unchanged. The behavior of redisplay on a particular zone depends on the exact subclass of the zone.
- Update the visible area of the screen according to the new zone hierarchy.

Chapter 7

Zone subclasses

7.1 Moving and resizing zones

A zone cannot move, so to obtain the effect of a moving zone, we use the following trick: A zone d that should be moved is made the single child of a zone D. To “move” d, replace D by D' which is like D but with a different position. The same method is used to “change the size” of a zone.

7.2 Layout zones

7.3 Application frame

An application frame is a layout zone with a small number of fixed layouts of sub-zones (we don’t call them child zones because they are not the direct children of the application frame. Selecting a particular layout has the effect of creating child zones of the application frame with positions and sizes that are determined by the layout, and each child contains a sub-zone.

7.4 Table zone

7.5 Simple application zone

This is a zone that executes a display function each time around the command loop in order to generate a complete hierarchy of zones.

7.6 Incremental application zone

This is a zone that is able to compare existing child zones with new ones, and that will recursively generate zones only if a child is new.

Chapter 8

Comparing two sequences of child zones

Initialize `before` and `after` to the beginning of the sequence of children *before* and *after* redisplay respectively. If `before` is empty, then all remaining children on `after` are inserted. If `after` is empty, then all remaining children on `before` are deleted. Otherwise (neither `before` nor `after` is empty), if the first zone on `after` is new, then it has been inserted; advance `after`. If not, the first zone on `after` is also in the `before` sequence. In that case, if it is also the first element of `before`, then it is either modified or unchanged according to the flag; advance both `before` and `after`. The last case is when the first element of `after` is not the same as the first element of `before`. Then the first element of `before` has been deleted; advance `before`.