

# The Hacker's Guide to the CDL (Cavor Display List)

by Alastair J.W. Mayer  
October, 2000

## ***Introduction***

The CDL (or more precisely, the CDLM -- Cavor Display List Manager) provides (vector) graphics display functions, including multiple individually selectable/viewable layers, multiple views (multiple windows into different areas of the same graphic), a degree of device independence, and other features that the typical "canvas widget" doesn't. If you're reading this, then presumably you have some idea of what Cavor is (or at least hopes to be): a general purpose tool on which applications which combine attribute and graphic data, such as GIS or CAD, can be built. It's also hoped that the various components that go into Cavor will find use in other projects (and perhaps vice versa). The goal of this document is to provide enough of an overview for those interested to either enhance the CDL in the Cavor context or to use it within another project.

The CDLM consists of a couple of other parts beyond the CDL itself. There's the interface layer that lets the Display Manager act as an independent process, and there's also the "mouse" process which provides for user graphic manipulation of the display, such as selecting features, digitizing points, lines and areas, and so on. (The process is somewhat misnamed, it will also (in the future) allow interaction via a digitizing tablet or other pointing device -- perhaps "pointer" would be a better name.)

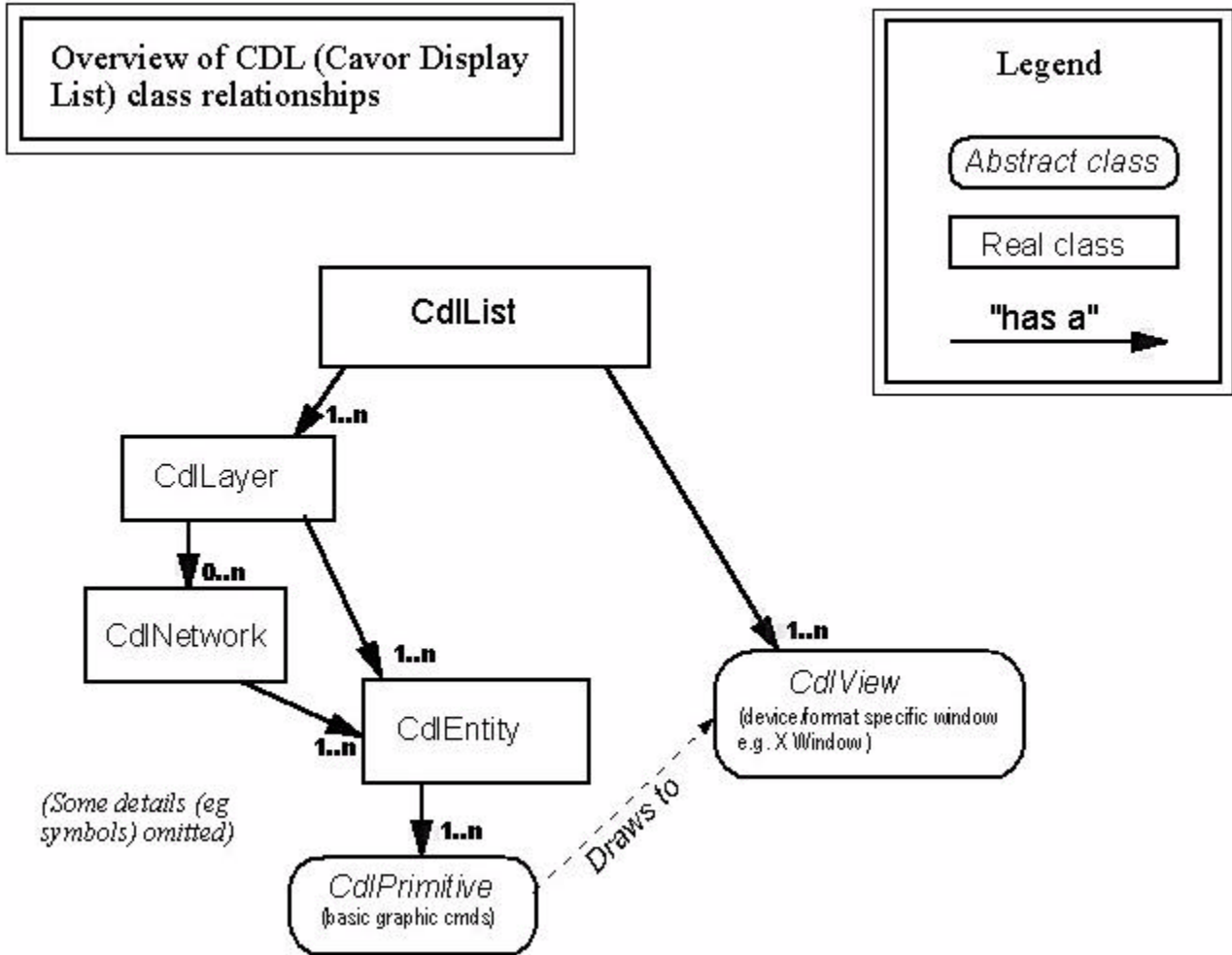
## ***CDL - The list***

A "display list" is a structured (usually) list of instructions for creating a picture on a graphical display. Vector drawing programs (such as Corel Draw, xfig, KIllustrator, etc.) typically use display lists; they allow selective editing and display of different components (lines, symbols, polygons) of the final image, as compared to raster editing or "paint" programs (such as the GIMP, Adobe Photoshop, etc) which operate on pixels.

The Cavor Display List provides structures parallel to the typical applications for which it is intended. A list (and the CDLM can actually handle multiple lists) or display contains a series of "layers". In a mapping application, one layer might contain roads and streets, another layer would contain water features (rivers, lakes), and another might contain graphics representing political boundaries (city limits, county lines, etc.). The user can turn on and off layers separately to unclutter the display. Each layer can contain multiple "entities" (features), which may (as in the mapping case) represent some real-world thing or might represent something more abstract (e.g. a class object in a UML diagram).

Cavor also adds the concept of "networks". A network is similar to a layer in that it is a

collection of similar or related entities, but it has additional properties more related to the properties of the entities in it. For example, electrical cable features in a map might be grouped into an electrical network which provides (through attributes on the database side) for analysis of connectivity from one point to another. For purely graphical applications, networks can be largely ignored.



Each entity consists of a number of "drawing primitives", the actual instructions as to how to draw the picture representing the entity. For example, a river entity might be drawn using two primitives: "color(blue)" and "line(pointlist)". Cavor currently includes primitives for setting the drawing color, line style (eg solid, dashed, etc), line width (thickness), and for drawing lines, polygons, text, and defined symbols. (A "symbol" is in some senses a macro defining a list of primitives.)

The list API includes methods to create (add) and delete layers, networks, entities and primitives. There is a concept of the "current" layer, network and entity -- adding a new entity defaults to putting it in the current layer and network, adding a primitive defaults to the current entity, and so on. All displayables (layers, networks and entities) have an id number by which they can be referenced or located.

Of course, these structures are of limited use in themselves, they need somewhere to display to.

## CDL - The views

Most applications that use display lists stick to a "one list, one view" paradigm -- you get one window onto the virtual graphic represented by the list, although that window may be zoomed in or out, panned around, etc. Cavor provides the option of multiple views per list: you can have one window zoomed in to, say, the NE corner of a map, another zoomed in to the SW corner, and perhaps a third showing an overview of the whole thing. Substitute "drawing" for "map" in other sorts of applications. Each view object keeps track of its extents (the area of the overall list that it covers), the scale it is displaying at, and so forth.

Views also provide a level of device independence. The `CdlView` class contains the (virtual) methods that the primitives use to actually display themselves, and specific implementations can provide for output to different types of devices. The most frequently used implementation is of course for X Windows, although this might be further subclassed where different display properties are an issue. Cavor also includes a PostScript view. This actually "displays" to a file which may then be printed, although it could potentially be extended to a full-fledged Display PostScript view class for use on systems that support DPS. Other possibilities include adding classes to support plotters, other file formats (e.g. SVG for the web), and the like. (The advantage to making these subclasses of `CdlView` rather than creating a separate export/translation facility is that it lets you convert just the area of interest defined by the view.)

## Using CDL - an annotated example

The following is an annotated example of using CDL linked directly into a program (via `libCdl`) rather than using the CLDI interface. For an example of the latter, see the program "cdliMapTest.C" in the `src/dli/` directory of the Cavor source distribution.

This sample program (it's from "test2.C" from the `src/cdl/tests` directory) creates a list, opens two views, and creates a couple of simple lines and some text in the display.

*Skipping the #includes, see the actual source for those.*

```

/*****8
* MAIN
*****/
int main(int argc, char **argv)
{
    char mousecmd[50];

```

*Do some basic initialization. (The parameter is the highlight color number, but that isn't working well right now.)*

```
    cdlInitialize(255);
```

*Open a display list. If it's already been created in this session, just activate it, otherwise created it. The 0L is the list ID, the next two parameters are the background and "dim highlight" colors to use.*

```
    cdlOpenList(0L, 0, 100);
```

Set the world coordinates of the lower-left (25, 25) and upper-right (300, 200) corners of the list display area. This has nothing to do with window coordinates.

```
current.list->set_wc(dbl_extent(25.0, 25.0, 300.0, 200.0));
```

Create an object of type `CdlXWindow`, a subclass of view designed to display on X11, and add that as a view to the list.

```
current.list->addView((CdlView*)
                    cdlOpenXWindow(":0.0", "wname", 0, 0, 50, 50, 300, 200,
0));
```

Start up 'mouse', passing the window ID as an argument.

```
sprintf(mousecmd, "mouse 0x%x &\n", current.view->window());
system(mousecmd);
```

```
dbl_extent ext;
int a = 550, b=350;
```

Make sure the view (view number 0) is zoomed out. Not strictly necessary since that's the default when created.

```
current.list->zoom_out(0, &ext, &a, &b);
```

Open (activate) layer number 1. If it doesn't exist in the list yet, it'll be created.

```
current.list->open_layer(1);
```

Add a new entity, ID number 101, to layer 1, network 0, and don't look for an existing entity of that number. If layer or network numbers are given as -1, then use whatever layer/network is currently open, otherwise open the ones requested.

```
current.list->addEntity(101L, 1L, 0L, 0L);
```

In the current entity (# 101, above) add a "text" primitive. The text string is "some text!", place it at position (50,75), with letters sized 10x20, at a 30.0 degree angle, using color# 50, align the position to the bottom left of the text.

```
current.list->text(dbl_coord(50,75), dbl_coord(10,20), 30.0, 50,
                CDLI_TA_LEFT, CDLI_TA_BOTTOM, "some text!");
```

Add another X Window view, this one a bit smaller. It'll be view #1.

```
current.list->addView((CdlView*)cdlOpenXWindow(":0.0", "wname2", 0, 1, 50, 50,
150, 100, 0));
current.list->zoom_out(1, &ext, &a, &b);
```

Create a PostScript "view", output into file "test2.ps".

```
CdlPostscript* psview = new CdlPostscript(3, "test2.ps");
psview->open("test2 ps file", 50, 50, 450, 300);
current.list->addView((CdlView*)psview);
```

*Add another entity (#102).*

```
current.list->addEntity(102L, 1L, 0L, 0L);
```

*Define some color numbers. (Note, the color model currently needs work; either the X server on my development machines is limited (cheap graphics cards) or I'm more confused than I thought about X11 color models. Or both.)*

```
current.list->defColor(7, 0xFFFF, 0xFFFF, 0);
current.list->defColor(9, 0, 0x8FFF, 0x8FFF);
```

*Set the line color to cyan.*

```
current.list->lineColor(9); // cyan
```

*Move to (100, 150) before drawing.*

```
current.list->move(dbl_coord(100, 150));
```

Set the linewidth to 3.0 mm on the display (absolute). A relative line width will scale with zooming in or out.

*Set the width of the line to 3.0 mm on the display. Absolute means that it will draw at that width no matter the scaling of the view, relative scales with zooms.*

```
current.list->lineWidth(3.0, CDLI_LINE_ABSOLUTE);
```

*Draw the line from the current point ((100, 150) set above) to 250, 100.*

```
current.list->draw(dbl_coord(250, 100));
```

*Set the line style to dash-dot.*

```
current.list->lineStyle(CDLI_DASH_DOT);
```

*And so on...*

```
current.list->draw(dbl_coord(75, 75));
current.list->lineColor(7);
current.list->lineStyle(CDLI_SOLID);
```

*A width of 0.0 is "minimal width", i.e. one pixel at whatever scale.*

```
current.list->lineWidth(0.0, CDLI_LINE_ABSOLUTE);
current.list->draw(dbl_coord(150, 200));
```

*Done with that entity, add another one (#103). Adding an entity automatically closes the previously open/active one.*

```
current.list->addEntity(103L, 1L, 0L, 0L);
current.list->lineColor(9);
current.list->text(dbl_coord(150,100), dbl_coord(10,10), 0, 9,
                 CDLI_TA_LEFT, CDLI_TA_BOTTOM, "N E W");
```

*All done adding entities, close the current one.*

```
current.list->closeEntity();
```

*Redraw (refresh) all the views. This takes care of those views that may have been added after some entities were already in the list. A view number of -1 means all the views for that list, otherwise it refers to a specific view number.*

```
current.list->redraw(-1);
```

*Flush any pending output.*

```
current.list->flush();
```

*Get rid of the PostScript view. The destructor closes the file.*

```
delete psview;
```

*The following takes care of refreshing the window(s) after expose events, resizes, and so on. Mouse events (button clicks, etc) are handled by "mouse", not here.*

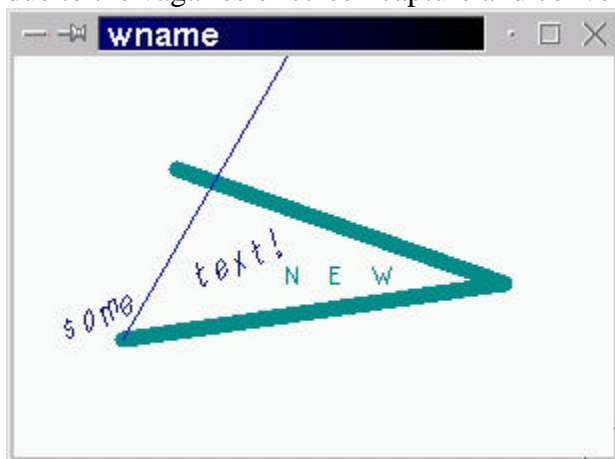
```

/*****
* Wait for X events
*****/
#define xEventFD() ((cvDisplay)?(ConnectionNumber(cvDisplay)):0)
    fd_set read_fds;
    int n_select = xEventFD()+1;
    while (1) {
        FD_ZERO(&read_fds);
        FD_SET(xEventFD(), &read_fds);
        select(n_select, &read_fds, NULL, NULL, 0);
        if (FD_ISSET(xEventFD(), &read_fds)) {
            cdlXEvent();
        }
    }
};
}

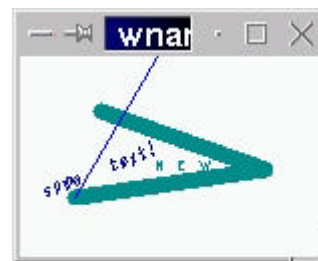
```

## Result

The resulting display (views 0 and 1) will look something like this (not a perfect reproduction due to the vagaries of screen capture and converting the image for this document):

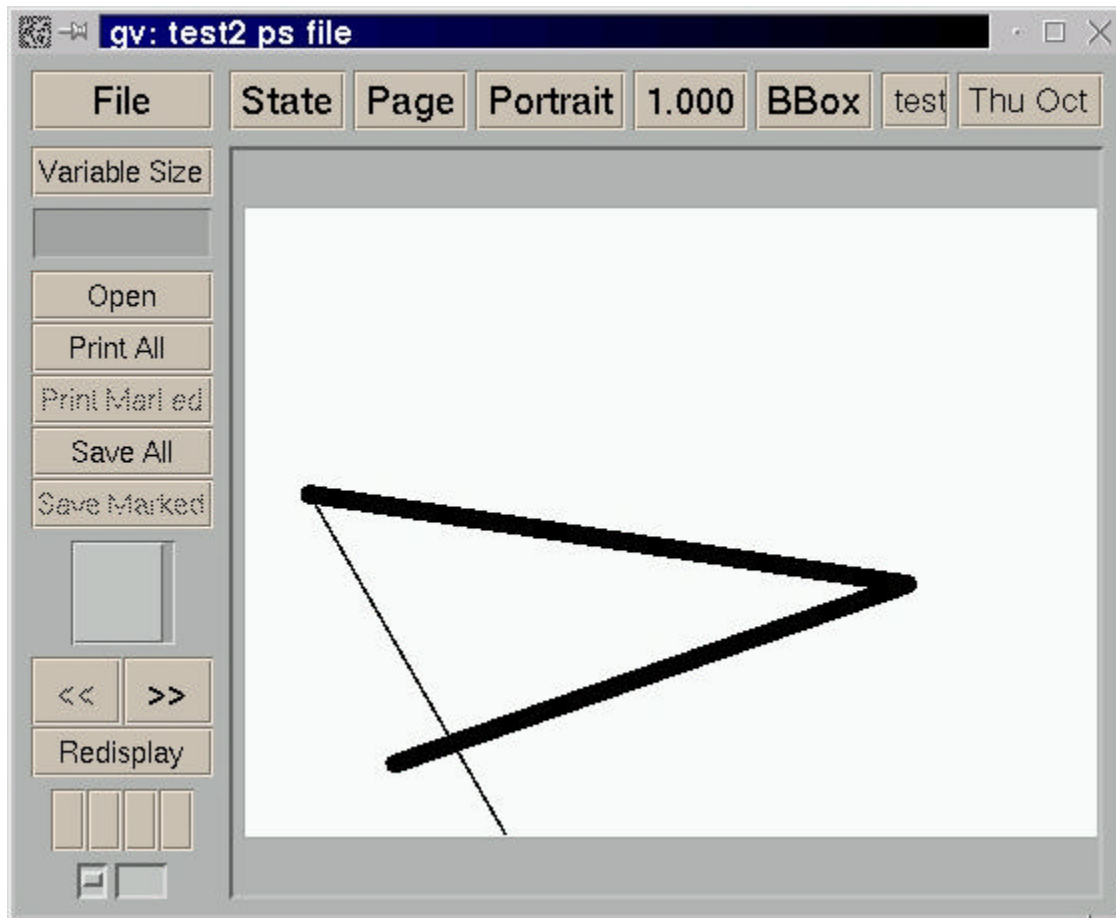


View 0



View 1

Observe that there are still some minor problems with text rotation -- the letters are slightly more rotated than the text string as a whole -- and with line style. The PostScript implementation is currently (October, 2000) incomplete; specifically it doesn't support color and the primitive used to display text is not fully implemented yet. (Also, the coordinate system is inverted since apparently PS uses a top-left origin rather than a bottom-left.) However, here's a snapshot of what the resultant PS file looks like in GhostView:



At least the lines work!

## ***Conclusion***

This has been a rather brief guide to CDL. You'll still have to dive in to the source for all the details, but this I hope this document is illuminating enough for you to get started understanding and using the code. Questions and comments can be addressed to the mailing list ([cavor-devel@lists.sourceforge.net](mailto:cavor-devel@lists.sourceforge.net)), you can subscribe to it and find other information at the Cavor project's SourceForge page ( <http://sourceforge.net/projects/cavor/> ) and the home page at ( <http://www.cavor.org/> ).

-- Alastair JW Mayer