22

Display Space Management

Though the Xerox Star[1] in 1981 was not the first instance of a graphical user interface with overlapping windows, it certainly was the most influential. It established in the mind of the research community the concept of rectangular windows scattered around a desktop each behaving like a smart piece of paper. It also established the software architecture ideas of model-view-controller and the operating system/graphics engine being the arbiter of screen space among competing applications.

It is hard to remember how computationally impoverished personal computing was in those days. Computers had 128K to 256K (not megabytes) of memory and 10 megabyte hard disks. Processing power was 8-10 MHz. Graphical displays generally used 1 bit per pixel. When the NeXT computer was introduced in 1988 its use of two bits per pixel was considered a little extravagant, though very nice to look at. Though personal computing has come a long ways, the rectangular overlapping windows have remained.

In this chapter we will look at some of the alternative proposals how to effectively use screen space that have become feasible with increased computing power. We will first address various styles for managing multiple rectangular windows on a screen. We will then look at the problems of how to deal with very large spaces that must be displayed in finite sized windows.

Window styles

There have been some studies of the ways in which windows are used and their effectiveness. Bly and Rosenberg[2] challenged the notion that overlapping windows are the most effective way to manage screen space. They predicted that when the relative sizes of window contents are known that a tiled strategy like that shown in figure 22.1 would reduce the user's window management burden The ViewPoint system (a descendent of Star) supported both overlapping and tiled windows.

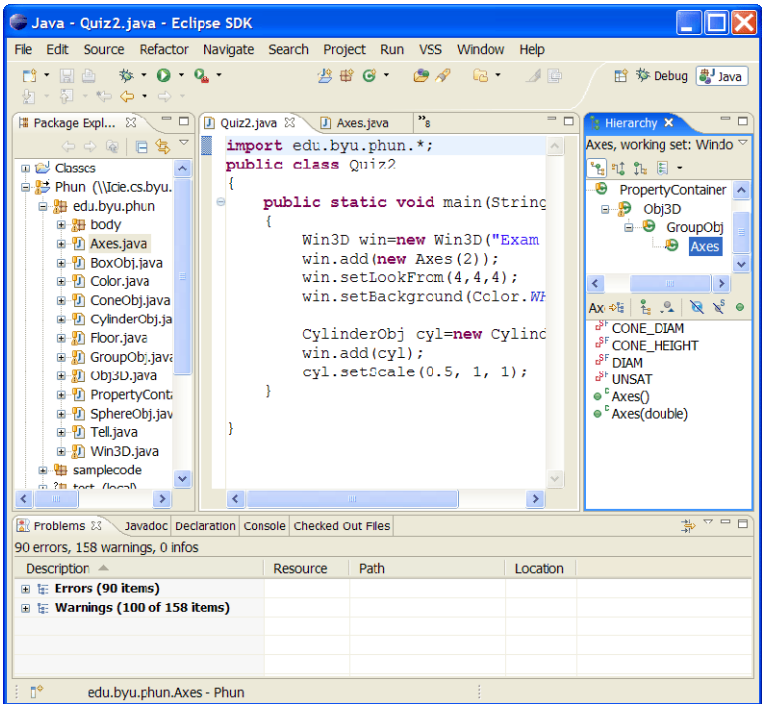

Figure 22.1 – Tiled windows in Eclipse

To evaluate window management schemes they created two tasks, one highly regular that would seem to conform with the tiled structure and the other irregular that would seem to conform better to the overlapping structure. For the regular task overlapping windows took 30% longer. The activity logs showed that users were performing twice as many window management tasks for overlapping windows than for tiled windows. For the irregular tasks the subjects fell into two groups. The "fast" group performed in one third less time with overlapping windows and the "slow" group performed in one third more time for overlapping windows. The key difference between the two groups was their familiarity with overlapping windows. This indicates that tasks requiring little window manipulation will be much more effective in a tiled arrangement that reduces that burden.

One of the assumptions of desktop window management is that windows will be placed over the top of each other with some windows completely obscured by others. Another assumption is that users will directly control the size of windows. These two assumptions are challenged in the Flatland[3] system for pen-based

interaction. In Flatland regions for drawing are called segments rather than windows. All open segments are visible at once. The currently selected segment is displayed at full size. The full size of a segment is determined by its content. Obviously not all segments can appear on the screen at full size. Flatland's approach is to shrink segments to make room for other segments. A user moves a segment by dragging one of the borders. Any segments that are in the way are moved to make room or shrunk. At some point movement or shrinking are not possible and a segment resists further modification. The algorithm is in figure 22.2. The Flatland approach to screen management is to have sizes and positions adjust automatically and predictably in response to movement of the selected window.

```
void move(Segment s, Rectangle newBounds)
{
    move s to newBounds
    foreach (Segment d that overlaps s)
    {   compute a location n for d so that it will move as little as possible an not overlap s
        move(d, n);
    }
    if (s overlaps any other segment or is outside of screen bounds)
    {   try to shrink s so that it will fit
        if ( s cannot shrink further and still overlaps a segment or exceeds bounds )
        {move s back towards its original position so that it does not overlap or go outside }
    }
}
```

Figure 22.2 – Flatland segment management algorithm

Figure 22.3 – Tilted and folded windows[4]

Several researchers have challenged the notion of windows being rectangular and only aligned with the X,Y axes. Beaudouin-Lafon[4] proposed that making windows more flexible would actually increase their usability. Figure 22.3 shows

windows that can be arranged at various angles and can be folded down. The tilting is controlled by the way in which windows are dragged around. This is modeled roughly on the physics of paper. Every window has a center. When one grabs a point to drag the paper, the direction of drag and the center of the paper tend to align. This is due to friction between the paper and the surface. The rotation to align, however, is not immediate because friction also inhibits rotation. The result of this simulation is that windows tend to behave much like paper being dragged across a desk with one's finger.

The usability advantage is that overlapping windows no longer align leaving fragments exposed where they can be more readily seen and selected by the user. Working with hidden windows is also supported in the folding technique. Grabbing the corner of a window and dragging it in towards the interior will cause the page to fold down like paper. This naturally reveals what is behind. This also folds any windows on top of the one being folded. This feels like leafing through a stack of papers as one might do on a normal desktop.

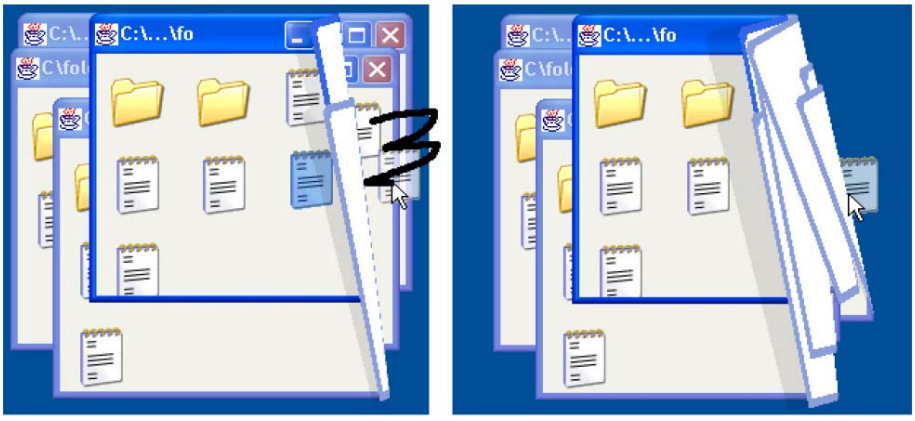

Figure 22.4 – Folding to drag and drop[5]

Dragicevic[5] expanded on the window folding idea to allow for more effective drag and drop on a crowded screen[5]. The problem is that to drag and drop requires that both source and destination be visible. Dragicevic uses mouse techniques coupled with window folding to simplify this. As shown in figure 4 one can pick up an object to be dragged and move it out of the source window. This causes the edge of the source window that was crossed to fold back temporarily. Crossing back and out again will cause the next window to fold and so on. One can use this "leafing through the pages" technique to reach the desired

destination, fold back the windows on top to expose it and then drop the object. Again this forms a natural simulation of paper behavior.

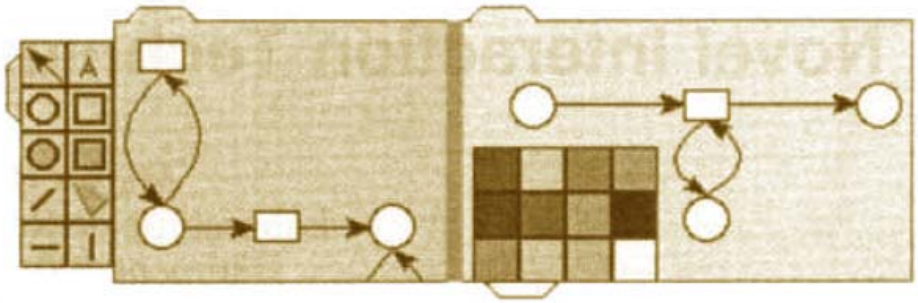

Figure 22.5 – Zipped windows[4]

Figure 22.5 shows Beaudouin-Lafon's windows being *zipped* together. This allows a group of windows to be treated as a single entity by the user. When one window (a slave) is brought next to another window (the master) they are zipped together at the common edge. Dragging the master window around will drag all of the other windows with it. Dragging the slaves will cause them to unzip and thus work independently. This allows user to dynamically group and reattach windows in meaningful ways.

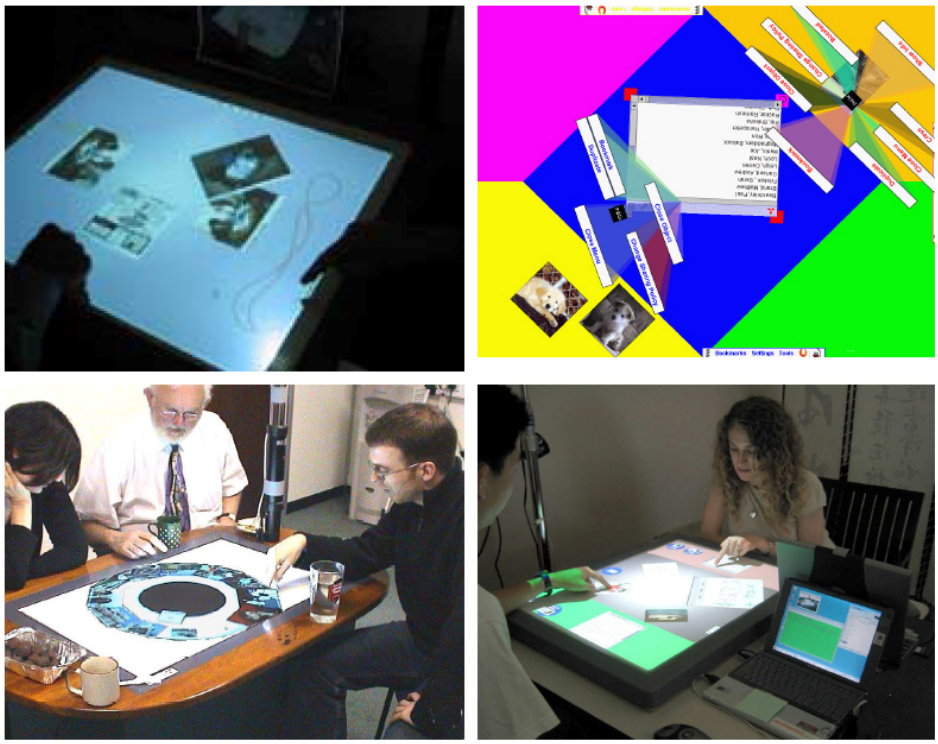

Figure 22.6 – DiamondSpin's polar coordinate windows[6]

The DiamondSpin[6] system supports people gathered around a table. This means that windows need to be oriented so that people can see them. Figure 22.6 shows this kind of usage. Because straight up and down images are inappropriate, DiamondSpin uses a polar coordinate system centered in the middle of the table for placing windows. In this system and window's orientation is static in the polar system, but as users drag windows around the screen their Cartesian orientation changes based on where the window is located in polar coordinates. The result is that windows end up appearing vertical to whoever the window is in front of.

Agarawala and Balakrishnan[7] pushed the physics-like ideas from Beaudouin-Lafon into increasing realism. Their BumpTop windowing system is supported by a physics simulator and a 3D graphics rendering engine. Windows have weight, friction and thickness allowing them to stacked, tossed, thrown around, folded and crumpled up as shown in figure 22.7. When windows are moved or

tossed they may strike other windows causing them to move. Their work also included innovative techniques for browsing through "piles" of windows. This included spreading them out like a deck cards and rummaging or leafing through the pile to examine what is there. BumpTop provides a lot of innovative ideas of how windows might behave. In the future we need data on which of these techniques may actually help users.

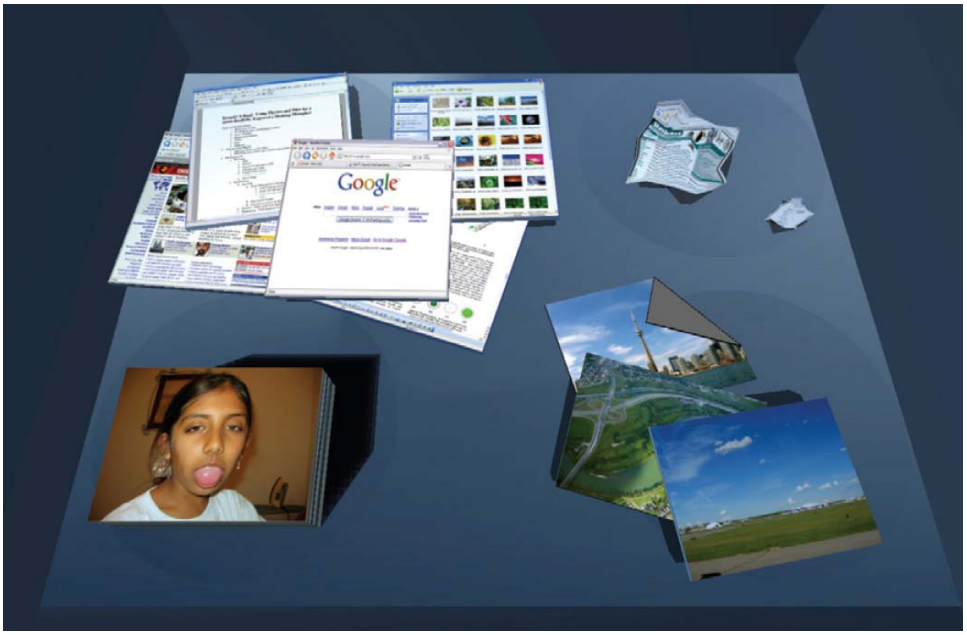

Figure 22.7 – BumpTop window physics[7]

The Scalable Fabric[8] system provides natural mechanisms for resizing and grouping windows. The screen space has a central focus area where windows are displayed in their full size and a periphery as shown in figure 22.8. When a window is dragged from the focus to the periphery it is scaled down to a much smaller size. The size is determined by the distance from the center of the screen. Windows are smaller when they are further from the center. Clicking on a window in the periphery will cause it to appear full scale at its original position in the central focus. Minimizing the window causes it to return to its peripheral position. In this mechanism windows never disappear or get obscured, they just get smaller or larger.

Moving two windows close to each other in the periphery will form a task or group, which can be given a name. Moving windows close to an existing task

will join that window to the task. This allows users to move related sets of windows around together both in the focus and the periphery.

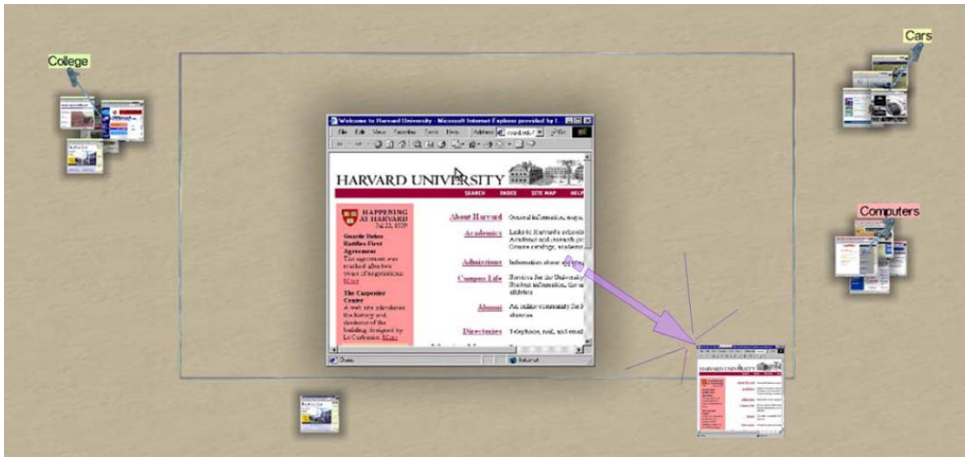

Figure 22.8 – Scalable Fabric[8]

Big worlds in small windows

The solutions described above all show ways in which the windows can be managed so as to simplify the use of limited screen space. In this section we will look at the problem of a single window that must view a much larger world space. The common solution to this problem is scrolling. In this section we will look at three alternatives: fisheye views, infinite zooming and focus + context.

Fisheye

A fisheye view is one where items at the focus of attention appear at full scale while more distant items appear with diminished detail the farther away they are from the focus. This concept was first introduced to user interface work by Furnas.[9] His work reported many instances of people using this diminishing detail approach when dealing with large amounts of information. When there is a large amount of material to be displayed there should be a "Degree of Interest" function applied to each item or region. His generalized Degree of Interest function was:

$$DOI_{fisheye}(x \mid .=y) = API(x) - D(x,y)$$

$DOI_{fisheye}(x \mid .=y)$ defines the degree of interest in point x given that point y is the current focus of attention. API(x) is the intrinsic global importance of point x

and D(x,y) is the distance between x and y. The API function can be many things. It can be uniform across the space, which will reflect all interest is around the focus of attention. In the case of a view of a map where the user is looking at Pendleton, Oregon (y) we might assign API(x) to be the population of each region x. Thus Seattle, Tacoma and Portland would have high DOI even though they are farther away from Pendleton. We could change API(x) to be the total number of software jobs in the area to produce a somewhat different map.

Though many fisheye views are geometric in their nature, such as maps, the generalized DOI can be applied to any information structure including trees, lists, tables and calendars. The fundamental idea is to allocate display resources where the interest is highest and to allow the user to change that allocation by moving their focus of attention. In a tree the API of a node might be the sum of the API values of each leaf and the distance might be the number of graph edges that must be traversed to reach one node from another.

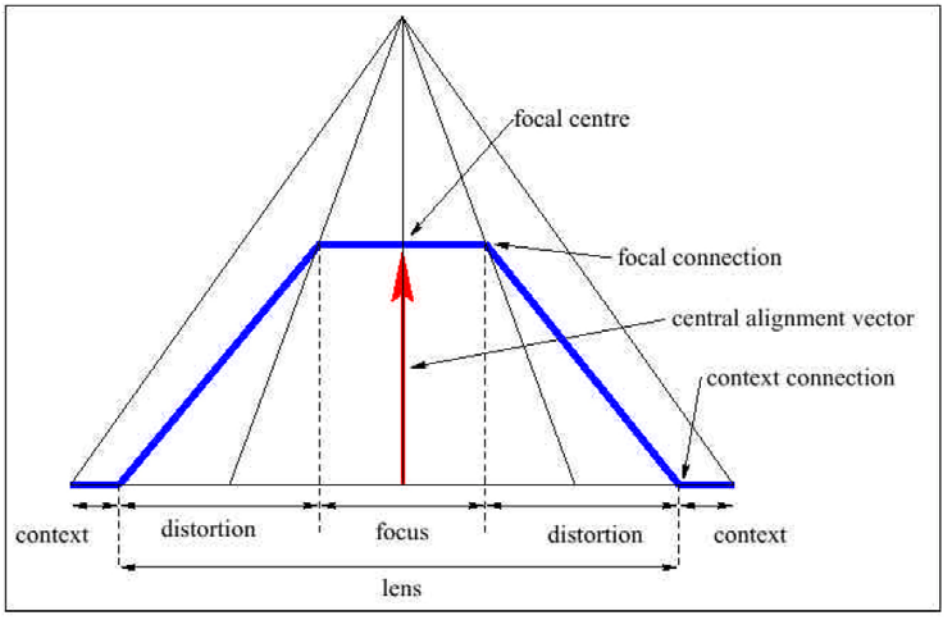

Figure 22.9 – Carpendale fisheye lens model[10]

Funas' seminal work on fisheye viewing has lead to many interactive techniques. Carpendale[10] created a space of possible fisheye views of a 2D plane. Her focus was on general techniques for transforming a large 2D space into a fisheye view within a limited space. Figure 22.9 shows the 1D representation of

her lens system for viewing. Figure 22.10 shows a sample fisheye view of a map. The lens model has a focus region that is undistorted and presented at full scale. The distortion regions allocate successively less detail as the distance increases and the context regions are a minimal scale and unaffected by the lens.

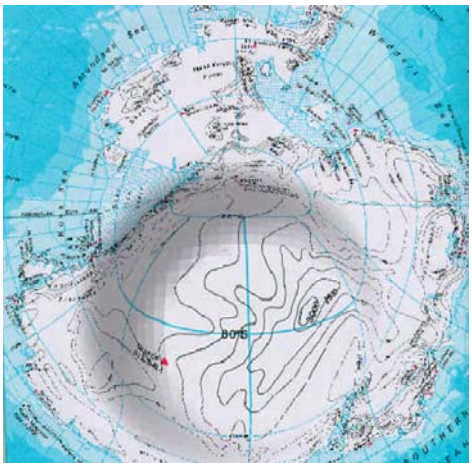

Figure 22.10 – Fisheye map view[10]

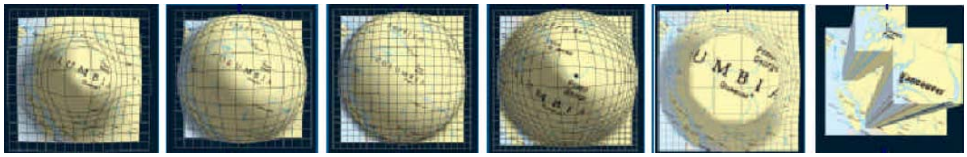

Figure 22.11 – Carpendale drop off (left to right) Gaussian, Cosine, Hemisphere, Linear, Inverse Cosine and Manhattan[10]

This model offers many parameters that can be adjusted to produce many viewing effects. Expanding the focus will show higher detail at the expense of the context area. The total width of the lens controls the amount of blending region possible between the focus and the context. A narrower lens produces a sharper transition. One of the most interesting effects is in the way that the distortion region drops off from focus to the context. Figure 22.11 shows 6 ways in which this drop off can be computed each producing various visual effects.

The actual fisheye viewing algorithm can be thought of as a blending between two transformations, *F* and *C*. Transformation *F* is the viewing transformation that transforms world coordinates into the coordinates of the focus region. This transformation consists of a scaling and a translation.

Transformation *C* transforms world coordinates into the coordinates of the context region. The context transformation also consists of scaling and translation and generally tries to scale the entire world into the available screen or window space. The drop off function *d(x,y)* produces values between 0 (context transform) and 1 (focus transform). It is based on the metric for measuring the distance between (x,y) and the focus of attention as well as one of the 6 drop off techniques from figure 22.11. The idea is that for each point (x,y) in the geometry of our model we compute the a new transformation *T* using the following function.

$$T = fisheye(x,y) = d(x,y)\,F + (1 - d(x,y))\,C$$

The result is a transformation *T* that can be applied to the point (x,y) to yield its position on the screen.

Though Carpendale's work focused on fisheye transformations of 2D worlds, Furnas proposed a generalized DOI function to be used in a variety of situations. Bederson used this idea in his fisheye menus[11]. The technique allows very long lists of things to be selected by altering the scale around the mouse pointer. Schaffer, et. al.[12] reference a large number of fisheye techniques on data structures and propose their own mechanism for viewing large 2D networks of connected nodes. Their ideas was to cluster graph nodes using some hierarchic clustering technique and then decide on whether to open or close a cluster based on how close it is to the center of attention.

Fisheye views provide a way to look closely at some part of a large space without loosing connection to the context. Gutwin and Skopik[13] studied the problem of interacting with data that is presented in a fisheye view. In particular they looked at steering tasks where the user must navigate a path that is longer than the focus region. Their experiments show that fisheye performance was significantly better than using a traditional panning technique. The reasons for the improved performance were the ability to see the entire path in the fisheye the fact that one need not manipulate the scrolling while pursing the path.

The fisheye techniques are based on Furnas' idea of focus plus context. The user works in a focus area but needs to retain context for that work. Most implementations of this idea assume that screen resolution is uniform. Baudisch et. al.[14] provide a novel alternative to this assumption. They created a focus plus context screen that has a high resolution working area surrounded by a low resolution context area. This was done by placing a white projection area around a high resolution screen. The focus information is displayed on the high

resolution focus screen and a projector is used to display low resolution context information around the periphery as shown in figure 22.12.

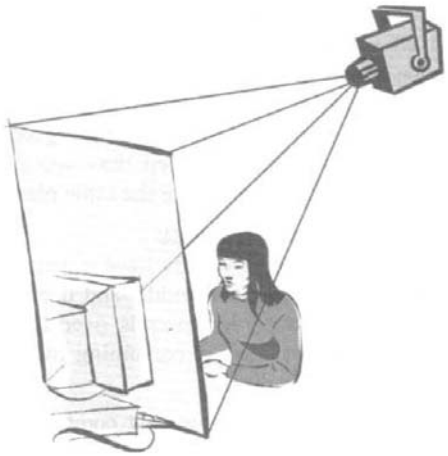

Figure 22.12 – Focus plus context screen[14]

Zooming

User interfaces have long had zooming as a mechanism for dealing with very large spaces. Zooming out provides lots of context. Zooming in provides greater detail. In 1993 Perlin proposed the Pad[15] model for organizing information. In the Pad model, the world consists of an infinite 2D plane. There are two kinds of objects on this plane, graphics (lines, circles, images, etc.) and portals. What is special in Pad is that this plane is also infinitely zoomable. That is the user can zoom in on any region to any degree of magnification. A portal is a region of the zoom space that references another region of the zoom space at any magnification or offset. Thus a portal can be used to link parts of the space together and to explore other parts of the space. Portals may also have filters that allow data to be viewed in a different way.

Pad introduced the concept of *semantic zooming*. This is where objects change their appearance as the user zooms closer. For example from long range a document is best presented as an icon with squiggles rather than actual text. However, as the user zooms closer the document should open and its pages should appear. As the user zooms even closer the actual page layout with editable text will appear. This editing level of detail is wasted when viewed from a great zoom distance.

The advantage of the Pad approach is that opening, closing, arranging, organizing, link following and many other interactive behaviors are all subsumed into a single model of panning and zooming across an infinite space. Because the space is infinitely zoomable there is always room for any activity because any region can contain an infinite amount of space when viewed at higher magnification.

The key technical challenge in zoomable interfaces is the representation of the infinite zoom space. In reality the space is not infinite because of maximum number precision. At the heart of the technique is the Pad *address A=(x,y,z)*. This includes x and y as well as a zoom location z. In essence a Pad address defines a linear transformation $T(u,v)=(x+u2^z,\ y+v2^z)$. Note that the z component is defined in the $\log_2$ space. This allows a number to reference a very large space of zoom factors. This is one of the keys to Pad's zoom space. A Pad *region* consists of an address and a width and a height. This is a rectangle defined at some zoom magnification. A portal exists at a region in the Pad space and in addition has a *look on region* which is the region to be viewed through the portal. Changing the look on region of a portal in essence changes the pan/zoom of the information viewed by the portal. At the top level Pad is viewed through a portal associated with Pad's window. Portals can view other portals and thus produce a recursive transformation sequence like those discussed in chapter 13.

Semantic zooming is achieved by giving every object (graphics or portal) a *visibility range* and a *transparency range*. When objects are rendered through a chain of portal views they are transformed to some level of magnification. The visibility range defines the range of magnification where that object is visible. If magnification is too low then detail views should disappear. If magnification is too high then summary views should disappear so that detail views can be visible. To create smooth transitions the transparency range defines magnifications where the object is only partially visible.

Perlin's original Pad implementation had efficiency problems. Many of these were resolved in Pad++[16]. The interesting question is whether pan/zoom techniques like those found in Pad are superior to the fisheye techniques described above. Nekrasovski et. al.[17] report extensively on experiments that address this question. Their conclusion was that zooming is superior to fisheye techniques. However, some care must be taken in generalizing their results because they are based on only one type of task.

[1] Johnson, J., Roberts, T. L., Smith, D. C., Irby, C. H., Beard, M. and Mackey, K., "The Xerox Star: A Retrospective," *IEEE Computer* (Sept 1989), pp 11-29.

[2] Bly, S. and Rosenberg, J., "A Comparison of Tiled and Overlapping Windows," *Human Factors in Computing Systems (CHI '86)*, ACM (1986), pp 101-106.

[3] Mynatt, E. D., Igarashi, T., Edwards, W. K., and LaMarca, A., "Flatland: New Dimensions in Office Whiteboards", *Human Factors in Computing Systems (CHI '99)*, ACM (1999), pp 346-353.

[4] Beaudouin-Lafon, M., "Novel Interaction Techniques for Overlapping Windows", *User Interface Software and Technology (UIST '01)*, ACM, (2001), pp 153-154.

[5] Dragicevic, P,. "Combining Crossing-Based and Paper-Based Interaction Pardigms for Dragging and Dropping Between Overlapping Windows", *User Interface Software and Technology (UIST '04)*, ACM, (2004), pp 193-196.

[6] Shen, C., Vernier, F. D., Forlines, C., and Ringel, R., "DiamondSpin: An Extensible Toolkit for Around-the-Table Interaction", *Human Factors in Computing Systems (CHI '04)*, ACM (2004), pp 167-174.

[7] Agrawala, A. and Balakrishnan, R., "Keepin' It Real: Pushing the Desktop Metaphor with Physics, Piles and the Pen", *Human Factors in Computing Systems (CHI '06)*, ACM (2006), pp 1283-1292.

[8] Robertson, G., Horvitz, E., Czerwinski, M., Baudisch, P., Hutchings, D., Meyers, B., Robbins, D., and Smith, G., "Scalable Fabric: Flexible Task Management", *Advanced Visual Interfaces (AVI '04)*, ACM, (2004), pp 85-89.

[9] Furnas, G. W., "Generalized Fisheye Views", *Human Factors in Computing Systems (CHI '86)*, ACM (1986), pp 16-23.

[10] Carpendale, M. S. T. and Montagnese, C., "A Framework for Unifying Presentation Space", *User Interface Software and Technology (UIST '01)*, ACM, (2001), pp 61-70.

[11] Bederson, B. B., "Fisheye Menus", *User Interface Software and Technology (UIST '00)*, ACM, (2000), pp 217-225.

[12] Schaffer, D., Zuo, Z., Greenberg, S., Bartram. L, Dill, J., Dubs, S., and Roseman, M., "Navigating Hierarchically Clustered Networks through Fisheye and Full-Zoom Methods", *ACM Transactions on Computer-Human Interaction*, 3(2), ACM(1996), pp 162-188.

[13] Gutwin, C., and Skopik, A., "Fisheye Views are Good for Large Steering Tasks", *Human Factors in Computing Systems (CHI '03)*, ACM (2003), pp 201-208.

[14] Baudisch, P., Good, N., and Stewart, P., "Focus Plus Context Screens: Combining Display Technology with Visualization Techniques", *User Interface Software and Technology (UIST 01)*, ACM (2001), pp. 31-40.

[15] Perlin, K., and Fox, D., "Pad: an Alternative Approach to the Computer Interface," *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*, ACM (1993), pp. 57-64.

[16] Bederson, B. B., and Hollan, J. D., "Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics", *User Interface Software and Technology (UIST 94)*, ACM (1994), pp. 17-26.

[17] Nekrasovski, D., Bodnar, A., McGrenere, J., Guimbretiere, F., and Munzner, T., "An Evaluation of Pan&Zoom and Rubber Sheet Navigation with and without an Overview", *Human Factors in Computing Systems (CHI '06)*, ACM (2006), pp 11-20.