The editor in HyperPascal is divided into three layers, each responsible for handling a particular aspect of the interaction with the user.

The architecture has been set up to allow for the existence of distorted views of the HyperPascal program. Using distortion-oriented techniques for presenting the program will allow us to counter one of the criticisms which is often levelled at visual programming languages. That is, that they are very wasteful of screen real estate. Now, it is possible to argue that textual programming languages, and common layout practice are wasteful; a significant amount of the text in most textual programs does not specify operations to be performed on data or (in typed languages) data types, but exists solely to disambiguate the structure of the program. Begins, ends, semicolons, then and else all fall into this category. Whenever they appear on a line on their own, a line is wasted. Similarly, space is wasted at the start of lines of text because of the common habit of indenting programs, and at the end of lines of text because new statements are normally started on a new line. This is not to say that indentation and the new-statement-on-a-new-line conventions are inappropriate. Far from it. They are important aspects of the programming technique. However, viewed purely from the point of view of space consumption, they are wasteful.

In spite of this argument, however, it must be acknowledged that visual representations of programs are never frugal in their use of screen real-estate. That is why we have planned for program display using distorted views, to allow a particular area of the program to be represented full-size, and surrounding areas to be reduced in size.

This has led to a three-layer architecture for the HyperPascal editor. At the layer, also called layer 1 or the concrete view layer are the objects with which the user interacts directly. These are distorted (the distortion will usually be 0 for the objects that the user is interacting with directly, but as a whole, they are in a distorted space, so they are, perforce, distorted). Furthermore, the screen may display more than one window onto the same region of the program, and the windows may use different distortion techniques, or different amounts of distortion.

This brings us to the reason for the second layer of the editor architecture, layer 2, called the abstract view layer. Layer two contains a representation of the undistorted graphic appearance of the entire program. Each of the concrete views at layer one is a window onto this canvas, and a distortion mapping operates between the two layers.

In some respects, layer 2 is similar to the base view developed by John Grundy in his MVIEWS system. However, the HyperPascal editor architecture also contains a third layer, the semantic view layer. Whereas the top two layers contain only graphical information about the program, the semantic view contains a representation of the program's functionality, expressed as a tree - essentially, a conventional parse tree. The nodes in this tree also reference a separate symbol table, but, to date, this has not been allocated a layer of its own. It can be regarded as another component of the semantic view.
Figure 1 shows the three-layered architecture, and the information that flows between the layers to maintain consistency between the parse tree and the screen representation of the program. When user actions occur in a concrete view, it is detected by the objects in the view, and they may respond with non-semantic feedback. For example, rubberbanding could be implemented at level 1, but there would be no semantic significance associated with the rubberbanding \textit{per se}. Level 1 also reports the user actions to level 2, in undistorted (i.e., Level 2) coordinates. Thus each level 1 distorted view is responsible for translating its own coordinates into undistorted coordinates (and also, as we shall see later, for translating undistorted coordinates into its own distorted coordinates). Level 2 does not, therefore, have to know anything about the distorted coordinate systems.

At level two, details of the physical structure of the screen objects are abstracted away. For the most part, at level 2, the information about each object will comprise an object identification, an origin and a size. It's not clear whether it would be better to store more information at level 2 for objects with special properties (like hot points); user operations on a hot point could be handled at level 1, and the abstract information relayed to level 2 (something like "user has clicked on hotspot of object n") but that would require level 1 objects in all level 1 windows to know about hotpoints. The other alternative would be simply to undistort the user actions at level 1 and report, for example "the user has clicked at (undistorted) coordinates (x,y)", and leave level 2 to sort out that this is in the hotspot of object n. The latter is more consistent with the divisions of responsibility for which the layering was intended. However, as the system is implemented in Delphi, and the Delphi objects will be at level 1, and are capable of detecting things like mouseclicks on hotpoints, it seems best to give this responsibility to level 1, and require level 1 to report to level 2 in terms of the higher level concepts (mouseclicks on hotpoints, object n has been dragged over object m, and so on) instead of the lower-level coordinates (coordinates where actions have occurred).

Level two must distinguish between actions which have semantic consequences and actions which have only graphical consequences. If the user drags an object to a new location which does not affect the meaning of the program, Level 2 simply instructs the

\footnote{Note that it is perfectly permissible to have a distorted window with a uniform distortion of 1 at all points; that is, an undistorted concrete view.}
level 1 representations to redraw themselves. If the user drags an object to a new location which does affect the meaning of the program, level 2 must inform level 3.

After the parse tree has been updated, Level 3 sends a message back to level 2 to redraw the program. Level 2 knows about the sizes of the various components, and consults the parse tree to discover their structural relationships. It redraws the expression based on this information.

For example, if the expression is \( x \ast (y+z) \), then the drawing comprises an expression box containing a left operand, an operator and a right operand. Actually producing the drawing involves a recursive traversal of the parse tree, filling in a standard graphical template for each expression in the tree.

The left operand goes at the top left of the template, and has whatever size is returned by the recursive traversal of its subtree. The left a-bar’s left edge is at the middle of the left operand’s x-width, and its top edge is offset from the top of the template by the height of the left operand. The right operand goes at the top of the template, and its left edge is offset from the left of the template by the width of the left operand plus a standard interoperand separation (say three characters). Its size is whatever size is returned by the recursive traversal of its subtree. The right a-bar’s right edge is aligned with the middle of the right operand’s x-width, and its top edge is offset downwards from the top of the template by the height of the right operand. The bottom edges of the left and right a-bars are a standard distance lower than the y-value of the top edge of the lower a-bar. Thus, although their top edges do not have to align, their bottom edges will. The operator’s x-location is mid-way between the maximum x-value of the left operand and the minimum x-value of the right operand. The operator’s top and bottom edges are equidistant above and below the bottom edge of the association bars. Not that asterisk is usually positioned as superscript, and it may be necessary to position it lower than the other characters to line it up correctly with the association bars. The size of the expression is \( \max_x \text{right operand}, \max_y \text{operator} \). If the expression is an operand in another expression, its size is used as the operand size in the parent expression.

Size information for the low-level components is stored at level 2. Size information for compound expressions is calculated anew each time the expression is redisplayed.

To summarise: user actions are captured at level 1 by Delphi objects, which generate a timestamped action history in terms of distorted coordinates. The action coordinates are undistorted at level 1 and sent to level 2. Level 2 notifies level 3 of any actions which may alter the parse tree (but remember the earlier footnote which suggested that all actions may be reported to level 3, to keep things simple). Level 3 updates the parse tree as necessary, and instructs level 2 to redisplay it. Level two constructs a new undistorted picture as described above, and instructs the level 1 windows to display it. They apply their own distortion functions to the graphical information and display it accordingly.

---

2An alternative might be for level 2 to tell level 3 about everything, and always get back a "redraw parse tree" message, whether the parse tree had had to be changed or not. Then it would redraw according to the information in the parse tree and its own information about object sizes. Sometimes the parse tree would have changed, causing an alteration in the structure of the graphical representation. Sometimes the sizes of objects would have changed, causing a change in the appearance of the graphical representation. And sometimes they would both have changed. This approach may be more consistent with the layering concept, as it does not require level 2 to know about things at level 3.

3Note that we can allow for a gap between the a-bars and the objects with which they are associated by drawing them so that they stop a standard distance short of their minimum bounding box.
Let us now consider the process of updating the parse tree.

Consider editing the sequence involved in updating an expression from \((x*y) + z\) to \(x*(y+z)\). The user simply drags the left association bar of the operator + into the vicinity of the operand y. The system's actions are a little more complex. Let us start with a picture of the situation before the user changes anything. At the top of the diagram is the representation which the user sees on the screen, with a grid superimposed over the expression to facilitate our understanding of the graphics. The grid does not appear in reality. The graphical units are arbitrary, and are not related to pixels on the screen.

Level 2 is represented in the upper grey area. It comprises a table of graphical information about each of the objects on display. Each of the objects is uniquely identified by a number, (see the diagram on the right) and the coordinates of its top left corner are recorded. It might be necessary to add size information into this table, but I've omitted it from this example because it isn't required. In addition, each object's entry in the table contains a pointer to its corresponding entry in the parse tree. Object 1, for example points to the location in the parse tree containing x. Strictly speaking, this entry only contains a pointer to the symbol table (grey area at the bottom of the diagram) in which the variable name is to be found. However, for clarity, the names of the identifiers have been included in the parse tree. Note that object 2 (the left association bar of the multiplication operator) points at a link in the parse tree, not a node. It is not possible to have pointers to links in the actual Delphi implementation, of course, but for the purposes of representing the manipulations which are occurring, it is convenient. An expression's entry in the parse tree may comprise a simple operand or a compound expression node. A compound expression node has a left pointer to a node representing its left operand, a centre pointer to a node representing its operand, and a right pointer to a node representing its right operand. Either of the left or right operands may be a simple operand or a compound expression.

The user manipulates the expression by dragging the left association bar of the + operator to the y operand. This sets in motion the following sequence of actions.

First, at level 1, the left association bar object redraws itself continually while it is being dragged around. Level 1 does not communicate with level 2 until the association bar is released in the vicinity of the operand y. Then the operand y sends a message saying that the association bar has been released in its vicinity. Level 2 does not update its graphical information at this stage, but informs Level 3 that this has occurred.
Level 3 updates the parse tree. In order to do this, it sets up five temporary pointers, *moving bar* (the one the user's dragging), *affected bar* (the one which will be displaced by the moving bar), *old operand* (the operand the moving bar was attached to), *new operand* (the operand the moving bar will attach to), and *old root* (a copy of the ordinary *root* pointer). Level 3 then points *moving bar* at *new operand*, *affected bar* at *old root*, and *root* at old operand. The result of all of this is the rearranged parse tree shown in the diagram on the right. Note that at this stage, nothing has changed in the table at level 2; the information recorded there about the graphical positions of the interface objects has not altered. At level 1, the representation shows the last position for the moving bar created by the level 1 immediate feedback mechanism. This is a temporary representation which is not consistent with the graphical syntax of HyperPascal, the information in the table at level 2 or the parse tree. This inconsistency will be resolved when the diagram is redrawn.
Redrawing the diagram involves passing information up through the layers again. Level 3 simply tells level 2 to redraw the diagram, and level 2 fills out its expression template by recursively traversing the parse tree, as was previously explained. This enables it to rebuild its table of (x,y) coordinates, and then send a redraw request to the concrete views at level 1. At level 1, the concrete views consult the table of undistorted coordinates at level two, map them to distorted coordinates, and redraw themselves accordingly.

The diagram at left shows the intermediate stage when the table at level 2 has been updated, but before the picture has been redrawn. The diagram at right shows the eventual result.

The preceding discussion has dealt mainly with the editing of expressions. However, the parse tree is not restricted to expressions, but also contains information about the structure of the whole program. Strictly speaking, the tree we have been discussing is a subtree within this larger tree, and the node we have been referring to as the root is only the root node for the subtree containing the expression which is currently being edited.

We have also ignored the possibility that the program may wish to make cosmetic alterations to the diagram, either within a single expression, or at a higher level (such as by repositioning a choice box or a loop node on the screen, without updating the meaning of the program. This positioning information is additional to information derived from the parse tree.

It is not clear exactly what information should be stored to represent cosmetic adjustments to the diagram for an expression, or the program generally. When a component is repositioned by being dragged to another position on the screen, is it being positioned further away from the component above, or closer to the component below? Is it being positioned so that the angle of its attachment wire matches that of another component, or so that the length of the attachment wire has a particular value? We could store the relative locations of the diagrammatic components in the parse tree so that when it is redrawn, it will appear with the correct graphical relationships between its components.

Another possibility would be to store the automatically calculated positions in the table as well as a displacement...
vector corresponding to the amount of movement which the user has applied to the object.