Draw Packages: A Survey and a Proposed Multitasking Implementation

by

Stephen Alan Williams

An essay
presented the the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1987.

ABSTRACT

A draw package is a computer program that allows a user to create accurate drawings in a structured format. This essay defines what a draw package is and how draw packages differ from programs such as paint systems and computer aided design systems. The differences between two dimensional and three dimensional draw packages are explained.

This survey outlines a representative collection of draw packages. A number of selected packages are introduced which, through various techniques, have increased the ability of a user to construct constrained drawings. Current trends in improved draw systems are listed.

A proposed and partially completed multitasking implementation of an interactive two dimensional draw package is described. The design is capable of producing high quality images in a format suitable for display on many output devices. The implementation is part of the Adagio workstation.

Acknowledgements

First of all, I would like to thank my supervisor, Kelly Booth, for his suggestions and for his patient reading of the many versions of this essay. Without his persistent prodding, I may never have completed.

Peter Tanner deserves a lot of credit for his patience and assistance, especially during the earlier stages of this essay. He acted as my unofficial supervisor prior to the writing stage.

I would like to thank Dave Fracchia, who acted as my student reader.

All the members of the Computer Graphics Laboratory at the University of Waterloo have given me some help during my stay here. Thanks also to all the friends I've made while at Waterloo who prevented me from doing my work and re-enforcing my opinion that there is life during and after graduate school.

Finally, I would like to thank my parents for all their encouragement and support (emotional and financial).

Table of Contents

Introduction

1.1. What is a Draw Package ?

A draw package is a computer program that allows a user to create accurate drawings in a structured format. These drawings may then be included in documents, used to create slides, or displayed on a video monitor.

The advantages of a computerized drawing system over hand drawing are many. The greatest of these is the ability to erase any part of a drawing at any time and as often as desired. There are physical limitations to erasing on a medium such as paper, but not on a computer screen. Another advantage arises when repetitive operations need to be performed. Computerized drawing packages frequently have built in shortcuts that make these operations easier and less error prone. Finally, the facility to interactively move picture parts around the screen improves the looks of the overall drawing.

Draw programs may either be interactive or language driven. Interactive packages usually consist of an image on a monitor that is modified in response to a user's manipulation of input devices. The draw program maintains a data structure representing the image at all times. A change in this structure results in an immediate change in the displayed image, and vice versa. By contrast, the image generated by a language driven package is controlled by a text file of drawing commands. A static visual image is created using this file as input to a program, which produces the image. Thus, any change to the definition of the image (i.e., a change to the text file) is not immediately seen. This essay is concerned with interactive drawing systems, and in particular with the multitasking implementation of a two dimensional package using multiple input devices in parallel. We will thus not delve deeply into language driven packages.

Draw packages are one example of illustration systems. An interactive illustration system can be placed into one of two categories: one which maintains a high-level description of a picture in terms of objects, and one which operates on raw frame buffer images. The first category includes draw packages, Computer Aided Design (CAD) systems, and videotex page creation systems, while the second category comprises what we will refer to generically as paint systems.

It would be hard to classify every interactive illustration package that maintains high-level descriptions of images as either a draw package or a CAD package since there is no concise definition of either. Instead, there is really a continuum between the two types of systems. CAD systems are generally more powerful and more application specific, while draw systems are less powerful but also more adaptable to different applications. Perhaps a more precise differentiation is that CAD packages are used for creating technical illustrations for use in engineering design and manufacturing, while draw packages are used to create illustrations to be included in documents. A further distinction is that CAD systems usually store semantic information in addition to that stored in draw packages. Examples are ancillary information such as the construction methods used to create an object and physical properties such as the electrical characteristics of objects.

Three construction techniques are currently used in computer illustration packages. Grid-based systems are easy to learn and to implement and have recently become popular. Constraint-based systems are also currently very popular due to their power, although often at the expense of useability. Recently, ruler and compass style construction systems have been introduced. They offer the advantages of both grid and constraint based systems.

Grid systems provide a constant reminder of the constraints imposed on a system, since they are often accompanied by a visible grid overlaying the drawing. However, no straightforward way is provided to express simple relationships such as specific angles or distances. When one wants to construct a line at forty-five degrees, he must count over and up an equal number of grid positions. Constructing a line at thirty degrees may be impossible on some systems. Grids are easy to use, but control is provided at too low a level.

Grids are in fact a very special case of a more general approach: constraint-based systems. Grids represent the constraint that all coordinates must lie on the grid. Other constraints, such as the angle of orientation or length of a line are also possible.

Essentially, there are three levels at which constraints can be applied to a drawing. The first and simplest form of constraint is forgotten as soon as it is used. For example, a user may specify that two lines be parallel. Here, one of the lines would be moved so as to be parallel with the second. However, these lines can subsequently be individually manipulated without affecting the other.

The second form of constraint is treated as an attribute of the individual drawing elements. Before each element is drawn, any constraints that apply to it are consulted. Thus, constraints are treated on an individual basis. Some drawing elements may not be able to be drawn to satisfy their constraints. This would occur if the constraints conflict with each other or with the constraints in other parts of the drawing.

The third, and most elaborate form of constraint, uses a system of constraints and a constraint solver. The system of constraints collectively describes all constraints on the drawing. The constraint solver is used in an attempt to simultaneously solve all the constraints within this system. It often does this by creating an algebraic equation for every constraint and attempts to solve the equations numerically. Sometimes the system is unable to be solved. Such systems will either have conflicting constraints, or have redundant constraints that the solver cannot manage.

Constraint-based systems are the most powerful of the three types of systems. However, this power is often difficult to harness. The user interface needed to present and control the constraints is nearly always complicated. Applying the proper constraints can be more like debugging code in a hit and miss fashion than graphical editing. After applying a number of constraints on a drawing, the user is still not sure if the proper drawing will be produced until he sees the final image. This may be caused by applying insufficient or redundant constraints. A user may have to experiment with various combinations of constraints before a final drawing is produced. An interactive constraint solver may produce unpredictable results and may drastically change the displayed image.

The ruler and compass approach has much of the power of constraint systems, but the user interface is simpler and the drawings are more predictable. Constraints are visible, as in grid systems, but in the form of visible construction lines and circles, much like the techniques used in high school geometry. The designers of these systems try to present operations in the same form that a draftsman would use. The drawbacks of this approach are that it may take considerable time to construct the constraints and that it is computationally expensive.

This chapter describes what a draw package is and the differences between two dimensional and three dimensional draw packages. The differences between a draw package and programs such as paint systems and CAD systems are also explained.

1.1.1. 2-D Draw Packages

The normal setup of an interactive draw package consists of the following: a video monitor on which an image is displayed; one or more input devices such as a keyboard, a mouse, buttons, switches, etc.; application software that controls the update of the image on the display, obtains data from the input devices and maintains information about what appears on the monitor; and of course a user, who manipulates the image with the input device(s). Many packages rely on only one or two input devices; either a tablet or mouse and a keyboard. The application software keeps some form of data structure to record what is on the screen and to enable it to make the necessary changes to the image in response to user input.

The type of image that can be created by a draw program varies considerably between packages. Drawings are constructed by invoking multiple operations consisting of parameterized primitives. For example, a line primitive can be parameterized by two end points, a point plus an angle with the horizontal, or a point plus the condition that the line must be tangent to a circle. Two dimensional packages have a subset of the following primitives available: points, lines, polygons, circles, arcs, and splines. In addition, some packages allow primitives to have modifiable attributes. Examples of allowable attributes include: colour, line type (e.g., solid, dashed, dotted), line thickness, degree of antialiasing, and the line end type (e.g., round, butted, square). Text of various sizes and fonts is also usually available.

Virtually all draw packages in existence have some form of menu-based control. The menus can be displayed on the screen along with the drawing, or defined implicitly as a collection of switches, buttons or other input devices.

The two styles of on-screen menus are fixed menus and arbitrarily displayed 'pop-up' menus. The fixed, or *static* menus are always present on the screen, and in some systems are able to be manipulated (i.e., translated, scaled). Pop-up menus, in contrast, are *dynamic* in nature. They only appear whenever a user requests that they appear. Often they are displayed at a location specified by the user (e.g., the tracker position), but they are not required to be. Once the selection(s) within a pop-up menu are picked, the menu is erased from the screen.

The screen is divided into areas for menus and an area to display the image. The options in these menus control scene composition such as additions and deletions of primitives, object selection and modification, transformations, and other picture building operations. Some systems allow for more than one window displaying the image to be shown at the same time. In two dimensions, the only uses for multiple windows are: to view the image with different attributes; to zoom, pan or scroll the image; and to operate on an individual object instead of on the entire image.

The image on the display is updated whenever a user action would cause it to change. Some systems will simply redraw the affected parts of the image without regard for the other parts of the drawing, while others will redraw the entire image for each modification. Some systems use two processors: one to modify the data structure and obtain user input, and another to periodically redraw the image using the data structure. These are usually referred to as the host processor and display processor, respectively. In the two processor situation, two copies of the data structure are usually kept: one that the application software uses on the host, and a 'display list' which contains purely geometrical information that is used by the display processor. A display list is arranged so as to minimise the time taken by the display processor to draw the image. While the display processor is traversing the display list, the host processor is free to modify the application data structure or perform other operations. In a few systems, the two processors share a single data structure. Here, no modifications to the data structure may be performed when the image is being drawn. The result is that the image cannot be updated as fast in this configuration.

A central component of the application is the model used for storing the data. This alone determines the structure and efficiency of the design procedures that must use it. For draw packages, this model usually contains some hierarchical description of the objects in the image. Some systems allow primitives to be clustered together and subsequently to be referred to as a single object. This hierarchy may be applied at higher levels by grouping together clusters. The model usually contains geometrical, topological, presentational and structural information. This is broken down as: geometric, the values of the coordinates of points; topological, which edges depend on which vertices; presentation, the locations of windows and menus plus any attributes of objects and display parameters; and structural, the definitions of objects and subobjects. In some systems more high level information such as relationships or constraints between objects is also kept. An example of this is the condition that line X be tangent to circle Y.

In addition to the operations of primitive addition and deletion, many more are potentially available. A user should be able to select any subset of the current drawing as the object for any command. This selection can then be copied to another location, transformed or deleted. Facilities should also exist to change the drawing attributes of a selection to other legal values.

Other common operations are:

- the ability to save the structure of an image onto some other medium, and later to recall it: Most often this involves saving a file of virtual coordinates in some standard format. The file can then be added to the existing application data structure at any time and can be converted into different formats. This makes the data compatible between different systems.
- the ability to change viewing parameters to affect operations such as zooming, panning and scrolling of the image: In order to manipulate sections of the image, they may have to be positioned at different locations on the screen or at different viewpoints.
- the ability to impose constraints on various picture elements. For example, a user may require that lines be parallel, perpendicular or equal in length.
- the ability to apply metrics to objects: A user would define a scale for the drawing, usually as a real world value (e.g., in centimetres). Lines could then be drawn to an exact length. Another metric that can be applied is the size of angles. Often, when metrics are allowed, provisions are made to allow the measurement of these quantities directly from a drawing.
- the ability to restructure the layout of the menus and windows on the screen: A user should be able to move the menus around the screen to his liking. These changes should then be saved so that each user could define his own environment.

- the ability to allow a limited amount of animation of objects: This would involve creating multiple images of the structure at different positions, and cycling through them.

1.1.2. 3-D Draw Packages

The features listed for two dimensional draw packages can be extended to three dimensions with some additions. The most noticeable difference is the need to use multiple views of an object for construction in three dimensions, so one is more likely to see multiple windows on the screen. Defining a point in three space with two dimensional input from devices using only one view is difficult and can be ambiguous. Therefore a user usually simultaneously interacts with the image from different views.

The same type of hierarchical data model that is used in two dimensions is also used here, except that a third coordinate value is stored for each point. This model is commonly referred to as the *line model* of three dimensional geometric modelling, in contrast to the models used in 3D CAD systems, which will be described in a later section. Again, only the coordinates of points and the lines which connect them are stored, along with some structural information. This can lead to ambiguity when one tries to define the surfaces of objects or tries to draw the objects using hidden surface removal instead of wire-framed. Because the line model cannot always define an interior and exterior volume for a given object, it is sometimes referred to as a 'two and a half dimensional model' as opposed to three dimensional.

All the primitive objects that were available in two dimensions are also available here. A three dimensional object is created by first drawing a two dimensional contour and then providing some depth or rotational cues. In this way the contour is swept over a volume, which then defines a solid object. A depth cue translates the contour in a given direction for a given distance. A rotational cue rotates the contour about an arbitrary axis (usually one of the primary axes). Another construction method is simply to connect lines between two planes in space. Some complicated geometric operations that are available in CAD require a great deal of calculation and are usually not included in three dimensional draw packages. For instance, the calculations involved in order to make two objects coplanar can get very involved and time consuming.

1.2. Draw versus CAD

A CAD system provides a number of tools intended to facilitate the work of a designer when creating, modifying or testing a product. The final design can then be used to control the operation of a number of machines in the production of a physical object. This final stage is referred to as Computer Aided Manufacturing (CAM).

1.2.1. In Two Dimensions

A CAD package that only works in two dimensions is used essentially for creating electrical circuits and technical drawings, logic simulation and cartography.

As an example of a 2D CAD package, an interactive editor of printed circuit boards will be described. A user may first define the size of the board to be used. Next, he may include components from a library, which can then be placed on the board. An important distinction should be made between draw and CAD packages. Both systems may allow the user to call up predefined objects, but CAD systems can be programmed to recognise many important and specific design rules and physical characteristics. For example, in this case the system would draw a chip at the proper scale on the board with the proper number of pins when it is referenced. After all components have been included on the board and the user has interactively joined them together, a few intelligent operations may then be available to him, depending on the system. First of all, there may exist a facility to optimise the current layout. The system would follow an algorithm to minimise a variable of the circuit (possibly the total length of connections between parts), while maintaining any physical constraints. The optimized solution is then displayed in place of the original solution. Another facility which would involve a lot of calculation, especially for large circuits, is a logic simulator. Given the physical characteristics of each component, the logic of the board could be compared to the desired logic to save time in building prototypes. Other possible operations that might be available in addition to those provided by draw packages include a bill of materials generation and area calculations.

1.2.2. In Three Dimensions

A number of the technical drawing software features can be useful for three dimensional modelling; in particular, interactivity, applicability and independence of hardware and dialogue methods. The types of primitives for three dimensional modelling include the standard two dimensional primitives, plus objects such as surfaces and solids [Gardan84]. Three dimensional CAD systems provide geometric modelling of objects in full 3D, in contrast to 3D draw systems. The models used in CAD allow for the definition of solids and an unambiguous description of what is interior or exterior to an object. In other words, the objects are represented from the point of view of their geometric properties. This permits a designer to perform operations such as structural analysis much more efficiently.

High quality geometric modelling must possess the following properties:

1) Any model that can be formed in the internal data structure must correspond to a real object.

2) A model of any real world object should be constructible.

3) Complicated geometric quantities such as volume may be calculated.

Sometimes it may not be possible to fulfill these properties using a line model representation. This may be due to a lack of information in the model or the fact that slow and complicated algorithms are required. For example, to calculate the volume of an object, its surfaces must first be known. With a line model it may be impossible or at least very difficult due to ambiguity to determine the set of surfaces comprising an object.

The above properties imply that some mathematical properties such as homogeneity, finiteness of dimensions, and rigidity should apply to the individual objects. These will ensure that any object created follows real world physical constraints. Homogeneity is the property that any solid object possesses internal solidity rather than empty space. Finiteness of dimensions simply means that all quantities that define an object must be finite and hence the volume occupied by the object is finite. Finally, rigidity constrains an object to have an unvarying form, no matter what position or orientation it assumes.

The processes that manipulate and alter the data model also require some guidelines. In particular, the following conditions must be met:

1) *coherence of operators* — that is, whenever an operation is applied to a solid object, it must produce another solid object.

2) *description* — any solid must be able to be represented.

3) *coherence of information* — a given point in space can only be contained within one solid object.

Three types of three dimensional models can be distinguished [Requicha80]. The first, the *line model*, which manages visual display-type elements, has already been described in a previous section.

The second model is referred to as the *surface model* or the *Boundary representation (B-rep)*. Here, definitions of surfaces are allowed which together may define a solid object. However, this form of model still may be insufficient to present an object unambiguously, since one still has to determine which of an object's surfaces define its volume. Surfaces may be defined as planes, rotated surfaces, lined surfaces, and mathematical models for surface approximations. There are limitations to this model in that there is no guarantee that any object can be produced and measurement of quantities such as volume can be difficult if not impossible to calculate.

The third and most complete model is the *solid model* or *Constructive Solid Geometry (CSG)* [Laidlaw86]. It can be used to represent complex objects and provide coherent information, particularly for object recognition. An object's volume can be enclosed by complex surfaces. There are two main methods used in solid modelling. The first recognizes an object by its silhouette. Topological and geometric information is stored as in the line model, but surfaces are known and oriented so as to define an interior and exterior. The second method recognizes an object in terms of the construction methods used to define it. Information is stored in the form of a construction tree in which non-terminal nodes represent operators, leaves represent basic objects, and the root is the generated object. An object is built by climbing the tree from the leaves to the root, defining subobjects along the way. A subobject is created at every level of the tree whenever an operator node is encountered. An example of a simple construction tree is depicted in Figure 1.1.

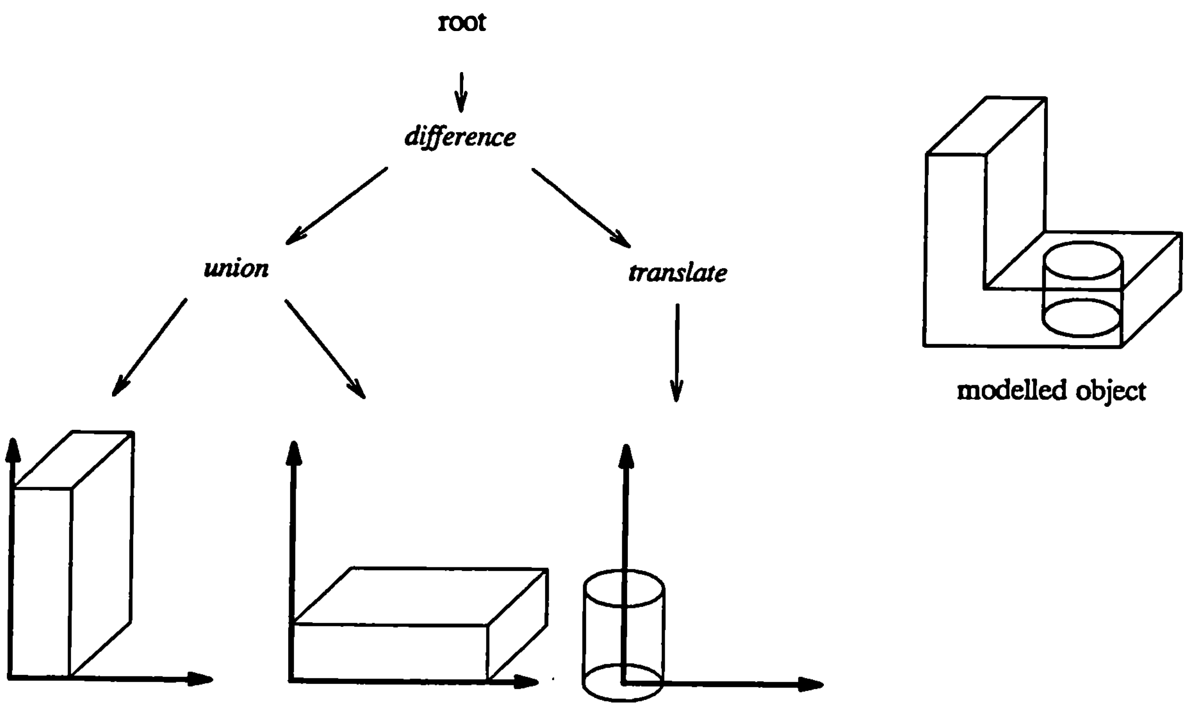

Figure 1.1 – A simple construction tree

Basic objects can consist of boxes, cones, spheres, cylinders, etc. The CAD software is able to combine these objects using Boolean operators such as difference and union. The solid model can meet the requirements of a high quality model in that any object can theoretically be represented, a data structure will correspond to a real object, and quantities such as volumes may easily be measured. A difficulty with this approach is that it is sometimes hard to transform the geometric model to a lower level graphical model suitable for use by a display processor.

Many of the methods used to construct objects on a 3D CAD system have been mentioned previously. To recap, these methods are: two dimensional drawing with multiple views, extrusion, and Boolean operations on basic objects. Extrusion involves using depth, rotation, contour joining and pierced depth along a path.

Generally, models are defined in such a way that the most frequent operations can be carried out easily. These operations can be very costly. For example, to generate an image with hidden surfaces removed requires a complicated algorithm, as does creating a cross sectional view. Another expensive operation is that of generating full-colour shaded images with hidden surface removal.

1.3. Draw versus Paint

For comparison purposes, this section will define a draw program to be any interactive illustration system that maintains a description of the displayed image at a level higher than at a per pixel basis. In contrast to draw programs, a paint system would not maintain a data structure of all the drawing primitives in an image for picture definition and manipulation. One may think of paint programs as simulating painting and draw programs as simulating drafting.

While the user interface issues of these two types of packages are very similar, there are noticeable differences. First of all, a 'select' operation in a paint program is able to specify a rectangular area of pixels which a user can manipulate (e.g., translate, copy, scale, etc.). In contrast, a select or 'pick' operation in a draw program would select one or more drawing primitives if they happen to be within some predefined pick window.

Secondly, a 'fill' operation in a paint program will start at a seed point of a given colour and will colour an area outwards in all directions until a pixel is encountered whose colour differs significantly from the seed point. A draw program will first determine if the seed point for the pick is bounded by a geometric object. If it is, the interior of this object is scan converted.

Lastly, a 'zoom' operation is normally handled differently between paint and draw programs. A paint program relies on either a hardware zoom facility in which each pixel is replicated according to the zoom factor, or a low-level software routine to magnify an area of pixels. This allows easy modification of individual pixels at larger zoom factors. In this scheme, a line of width two pixels when zoomed by a factor of two will appear four pixels wide. A draw program would normally rely on some form of software zoom to achieve magnification. This could be done by changing viewing parameters (such as the distance from the eye to the viewing surface) or resizing windows. Under this scheme, a line of width two pixels will always be drawn at the same width unless line widths are scaled in proportion to the zoom factor.

In general, there are a number of differences in the functionality of the two types of systems. The most apparent difference is that transformations may be applied to a group of objects in draw systems, whereas these operations are not easily defined in paint systems (other than translate and possibly scale). For instance, an arbitrary rotation of a grid of pixels onto another grid is difficult.

A second major difference is the ability to draw free form lines or curves. Whereas this is an essential feature of a paint program, it is rarely included in a draw package. A paint program can handle this operation by simply writing into each pixel that the screen tracker crosses. However, a draw program would have to represent the path as a collection of points, which is not easily manipulated or effectively stored.

A third difference may be the variety of brush styles available. Draw programs often have lines of varying thicknesses and on/off patterns (e.g., dashed, dotted, etc.), but all with a rectangular profile. Paint programs may or may not have these line types, but they often have arbitrary two dimensional figures for their profile and may allow for techniques such as air brushing [Smith78].

There are also differences which may appear transparent to a user. First of all, the image displayed in a paint system is generated by sending the contents of the frame buffer memory to the video controller. A draw system usually incorporates a display list as an extra step in this process. Every time the display list is updated, it is traversed to generate an image in frame buffer memory. When the current image is saved in a paint system, the entire frame buffer must be saved. Draw systems only have to save a form of data structure which represents the image. Therefore, the output from a draw program will be more portable across different systems than output from paint programs, since draw output is normally in virtual coordinates while paint output depends on the resolution of the viewing device.

Another difference is in the manner that undo operations are handled. For a paint system to allow an undo operation, a copy of the image before the last operation must be saved. This requires a lot of memory, so in most cases only one level of undo is provided or none at all. Draw systems can keep track of any number of previous operations made on the data structure. Since all the operations available are well defined, the inverse of each operation may be generated. As examples, the inverse of drawing a line is deleting that line and the inverse of a scale by two is a scale by one half. Each time an undo operation is invoked, the inverse of the previous operation is performed on the data structure. Thus, any level of undo is theoretically possible.

The availability of various colours and of constraints varies widely among draw and paint programs. However, in general, a paint program should not need elaborate constrained drawing as a draw program would, because of the types of images each system is designed to produce. In addition, while paint programs often provide colours from the entire spectrum, draw programs normally need not offer as broad a range of colours.

Survey of Draw Programs

2.1. Introduction

In this chapter, a number of selected packages are introduced which, through various techniques, have increased the ability of a user to construct constrained drawings. The first section describes the Sketchpad system [Sutherland63]. Sketchpad is described in great detail because it was the first draw package introduced and incorporated many of the tools desirable in a draw package. This enables subsequent packages to be described by the ways in which they differ from Sketchpad. The next section outlines a representative collection of draw packages. Finally, current trends in improving drawing systems are listed.

2.2. Sketchpad - the first Draw Package

2.2.1. Description

The first draw package introduced in the literature was Ivan Sutherland's Sketchpad at MIT's Lincoln Laboratory [Sutherland63][Sutherland84]. This system is still a major source of reference for designers of new systems and is a landmark in interactive graphics. Many of Sketchpad's ideas have been incorporated into current systems. The three facilities of Sketchpad which describe its functionality are: a hierarchical subpicture capability, a constraint capability and a definition copying capability. The ability to store and recall images from backup storage is also important.

A Sketchpad user interacted with the image on a monitor by manipulating input devices such as a light pen, push buttons, switches, and knobs. He issued specific commands with a set of push buttons; generated position information and picked existing drawing parts on the screen with the light pen; turned functions on and off with an array of switches; and rotated and magnified parts of the image by turning knobs.

At the time Sketchpad was produced, the only primitive drawing operations available were straight line segments and circular arcs. However, a user could define arbitrary symbols from any collection of drawing primitives including previously defined symbols. Once a symbol was defined, subsequent changes to its definition were seen at once wherever that symbol appeared. Any number of these subpictures could be used in a drawing.

Atomic constraints were included in Sketchpad to make lines horizontal, vertical, parallel or perpendicular; to make points lie on lines or circles; to make symbols (numbers or text) appear upright, vertically above one another, or of equal size; and to relate symbols to drawing parts such as lines or points. Any combination of these basic relationships could be composed and defined as a single condition to be placed on an object (or objects). For example, a number representing the length of a line could be constrained to that line and displayed along with it. Then, if the length of the line was changed, the constrained number would change accordingly. A graphical depiction of the conditions could be displayed along with the drawing to enable erasure or modification of these conditions with the light pen. When constraints were applied, Sketchpad would move parts of the drawing around in an attempt to satisfy these conditions as completely as possible. By using the definition copying facility, any condition applied to an object could be copied and applied to another object. This made the constraint capability easy to use.

2.2.2. Interaction

This section describes many of the ways in which one could interact with the image on a monitor using Sketchpad. These functions were performed mostly with the light pen, although some were controlled by knobs.

In the normal drawing mode, feedback was provided by a process called *rubberbanding*. After having selected a starting position for a line or arc, an echo of a line (or curve) from the starting position to the current tracking position was drawn on the screen. When a closed polygon was drawn, this method was used to construct the edges, while a button was used to select where the vertices were placed. To close the polygon, the light pen was brought near the first vertex, where it would lock on to the end. To terminate the drawing, a sudden flick of the pen was used. This would cause the computer to lose its tracking of the light pens' motion.

The ability to lock on to existing points in an image was a result of the system storing explicit information about the topology of the drawing and using this information to construct *gravity fields* around visible objects on the screen. When the tracker came near a gravity field, it was drawn into it and ended up pointing at a control point or exactly on a curve.

At any time during the construction of a drawing, *instances* could be created or defined. An instance could be created out of any combination of lines, curves or other instances either on the screen or accessible from tape. Once this was done, the instance could be copied to other locations on the screen without duplicating the memory of the original instance. Subsequent modifications to the definition resulted in immediate changes to all instances in the drawing. Sketchpad also provided an operation to simply copy any group of objects. In this function, the memory of the source objects was physically duplicated so that any operations could be performed on it at will without changing the original objects.

Other techniques that were used when interacting with Sketchpad included:

- if a vertex of a polygon was moved, all connecting lines would follow the movement in a rubberband fashion
- if a new point was to be generated at the pen location, but the pen already pointed to an existing point, no new point was generated, but the existing point was used again
- if a point was being moved and a termination signal was given while aiming at another point, these two points were merged
- if two lines were made collinear, the resulting line would only refer to its two endpoints
- if a point was created on or moved onto a line or circle, constraints were automatically set up to indicate this condition
- if an object upon which other objects depended was deleted, the dependent objects were also deleted (e.g., if a point was deleted, all lines which terminate at that point were deleted)
- if two independent objects of the same type were merged, a single object of that type would result (e.g., if two points were merged, all lines which terminated at either point would now terminate at the resulting point)

One final tool that was of great use was the ability to magnify any portion of the screen. The user was able to specify a scale factor for the display to enlarge specific parts of the drawing. This made it possible to draw the fine details within the drawing.

2.2.3. Constraint Satisfaction

Once constraints were applied, Sketchpad had to try to satisfy them simultaneously, although sometimes this was not possible. Constraint satisfaction was first attempted with a one-pass method, which gave rapid results but often did not work for other than simple cases. A solution was most often obtained by using the slow but reliable relaxation method.

To implement the relaxation method, each of the constraints in the system needed to be defined numerically. Sketchpad achieved this by expressing each constraint as an equation in terms of error introduced into the system. For example, constraining two points to have the same x-coordinate would result in an equation involving the difference between the x-coordinates which expresses the error term. Equal weights were given to each constraint equation. The relaxation method would then try to minimize each error term by manipulating the variables in each equation. It did so as follows :

1. A variable was chosen
2. It was re-evaluated to reduce the total error introduced by all constraints in the system
3. Another variable was chosen and the process repeated.

Since each step made some net reduction in error, the error decreased monotonically and thus stability was assured.

The one-pass method worked as follows: suppose that one of the variables in the system of equations could be re-evaluated to satisfy all the constraints that apply to it, due to the fact that it had so few constraints. This variable was then considered 'free'. Now, all constraints which were applied to it were removed from consideration. If a single constraint constrained two variables, these variables were considered adjacent. The removal of the constraints applying to the free variable resulted in a number of adjacent variables becoming free. This removed further constraints from consideration, resulting in more free variables, and so on. Sketchpad would then re-evaluate all the free variables in reverse order, saving the first found free variable until last.

As an example of the one-pass method, five variables with a total of four constraints are depicted in Figure 2.1.

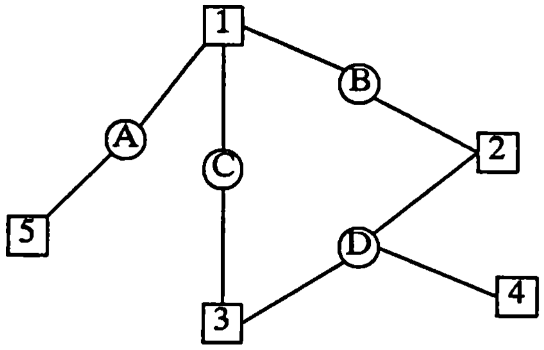

Figure 2.1 - A sample constraint diagram

In this diagram, variables are boxes labelled with numbers and constraints are circles labelled with letters. The variables affected by a constraint are those boxes that are connected by lines to the circle representing the constraint. To start off, variables 4 and 5 are likely candidates for free variables. Suppose 5 is picked as a free variable. Now constraint A is dropped from the list of constraints. No additional candidates for free variables are generated. Next 4 is picked as a free variable. This removes constraint D from the list and also makes variables 2 and 3 candidates for free variables. Variable 2 is picked which removes constraint B. Now, both 1 and 3 are candidates for free variables, and 3 is chosen. When constraint C is removed, no more constraints remain. The list of free variables is then re-evaluated in reverse order, that is, 3 first, then 2, 4 and 5. In re-evaluating a free variable, only those constraints which were present when the variable was found to be free are used.

2.2.4. Conclusions

Sketchpad was used in a number of applications including the definition of mechanical linkages, analyzing the force distribution in members of a bridge truss, and as an input to a circuit simulation package.

A shortcoming of Sketchpad, as described in Sutherland's thesis, was a lack of drawing primitives of different types. A spline drawing facility would have been useful instead of defining curves as combinations of circular arcs. However, Sketchpad was designed so that the addition of new drawing abilities would be straightforward.

Another shortcoming had to do with the graphical representation of constraints. A drawing containing many constraints tended to be confusing when they were displayed graphically, often overlapping one another. As well, each constraint type was represented by a single character inside a circle, called a tag. This tag may not have been sufficient to represent all of the different types of labels one would have liked to identify a constraint. However, considering the equipment that Sketchpad used, the functionality provided is still comparable to some systems in use today.

2.3. Later Draw Packages

2.3.1. Draw

Around 1975, Patrick Baudelaire at Xerox PARC introduced a system which he simply entitled 'Draw' [Baudelaire75][Baudelaire79]. His package was one of the first to introduce facilities such as multiple brush types and splines. It was also the first to introduce the concept of grid constraints. In addition, new concepts in user interfaces were introduced.

One major advantage Draw had over previous systems such as Sketchpad was the ability to draw with more than one style and size of brush. Draw provides four distinct types of brushes including dashed lines, dotted lines, dot-dashed lines and the default solid line. Each style could in turn be combined with four discrete line thicknesses for a combination of sixteen brush styles.

Draw also introduced splines curves to draw programs. The user first places each of the control points for the spline on the drawing surface. After this is done, Draw fits a spline through the control points. Any type of brush can be used for splines as well as lines.

Probably the most important development introduced by Draw is the ability to specify grid constraints. When invoked, an array of points at regular intervals both horizontally and vertically appears over the drawing surface. This array may then be scaled and/or translated to any desired position. The grid is used to constrain control points of drawing primitives to lie only on grid points. This enables the user to easily construct right angles, equal distance lines, and regular curves whose control points lie on the grid. Drawing rubberbanded lines in grid mode results in lines only being drawn when the screen tracker passes a grid point. If the tracker is positioned mid-way between two grid points, no line will be rubberbanded to the tracker; only lines at discrete multiples of grid points are ever drawn. A constraint may also be applied to restrict all lines to be either vertical or horizontal.

Draw improved the ability of the user to interact with the image on the screen. The two most noticeable improvements are the replacement of a light pen with a tablet or mouse and the display of fixed menus on the screen alongside the drawing surface. A light pen interferes with the user's view of the drawing surface and has a limited amount of coordinate resolution. Tablets and mice are able to provide much higher resolution and may also allow 'softswitches' if the device has multiple buttons. These switches could invoke default operations instead of having to explicitly make a selection from a menu.

Draw changes the screen tracking icon whenever a new operation is selected, providing feedback as to the semantics of the operation. Text is manipulated by selecting a string and moving it around with the tracker as one would any other object.

Draw has a novel approach to defining various transformations on an object. The normal transformations such as scaling, rotation and translation can be combined in the following way:

a) Select the object.

b) Define two reference points on this object by making selections when the tracker is above each point. These points will be marked with asterisks.

c) Select another two reference points anywhere on the screen. The selected object will be transformed so the original two reference points will be mapped to the second pair of reference points. The topology of the original drawing is preserved.

Non-traditional transformations such as stretching, slanting, and symmetry can be accomplished by an affine transformation that maps three source reference points to three target reference points. These transformations may be accomplished in fewer steps. For example, a scale may be described by anchoring a point, selecting another point, then dragging the second point to the desired location. Notice that the position of a fourth point is unnecessary. The choice between these two techniques for specifying a scale is a user interface consideration.

Draw is best known for its introduction of grid constraints. Although many of the newer systems use other constraint methods, they greatly resemble Draw in terms of user interface and functionality.

2.3.2. Star Graphics

At the Xerox Palo Alto Research Center (Xerox PARC), a great deal of research has been put into integrated document composition systems. Developments include Baudelaire's Draw, the Bravo editor [Lampson79], and the Tioga editing system which is part of the Cedar programming environment [Teitelman84]. Other systems include Newman's Markup and Stone's Griffin, which will be discussed in a later section. These systems were primarily research projects used internally at Xerox.

In the late seventies, Xerox developed the Star professional workstation [Lipkie82]. It was the first commercial system of this type developed at Xerox. The unique aspect of Star is the user interface of its integrated text and graphics editor. This section describes some aspects of that user interface.

The central concepts of the Star interface are 'What You See Is What You Get' (WYSIWYG) [Englebart68] and visibility. The visibility concept assures that the system does not hide things under "code plus key" combinations or force the user to remember many conventions. All actions should be immediately obvious to a user. Interaction is performed with a mouse and a keyboard. The keyboard is divided into two sections: the standard 'QWERTY' section, and a group of function keys which perform operations such as MOVE, COPY, DELETE, SHOW PROPERTIES and AGAIN. Their meaning is only defined once an object is selected. Grid constraints with redefinable grids are also provided along with their associated control.

An important part of an effective user interface is feedback. The Star editor provides this in a number of ways for object selection. Icons show that they have been selected by reverse video or highlighting. A selection of text is noted by highlighting a rectangular region around the characters. When a graphic element such as a line, point, or polygon is selected, each of its control points is highlighted by inverting a small square region around them. For text, multiple clicks of the mouse button have different meanings. The first click will select a character, the second a word, the third a sentence, and the fourth a paragraph. Clicking the other mouse button can then expand or shrink the selection at the current level.

In all of the above examples, a selection requires depressing and releasing the mouse button. This is an important distinction to make because a user may depress the button while pointing to one object and then move the tracker around.

An object is then selected if the button is released above it. Throughout this process, the editor will highlight which object is currently under the tracker, and hence the object that would be selected if the mouse button were to be released.

A very useful tool that was rare among draw packages around the time Star was introduced allows copying and changing properties of objects. This ability is much more common today. If an object is selected, the SHOW PROPERTIES function will open up a small window in which a *property sheet*, displaying the properties of the selected object, is placed. For instance, some properties shown for a line would be: line thickness, line style (e.g., solid, dashed, etc.), and any constraints applied to it. For each property, either an enumeration of all possible values or a box into which a value is to be typed is displayed. The current value applied for each property is highlighted and may be changed simply by selecting another value for that property. Another way to change the properties of a selected element is by the use of the COPY PROPERTIES function. To use this function, it is invoked and then the object from which properties are to be copied is picked. The properties of this picked object are then copied to the currently selected object.

Packages that have a 'copy properties' function usually only allow one property to be copied at a time. Usually the user has to determine the value of the property from the source object and manually change the target object's property to this value. This can be time consuming if this needs to be done for many objects or many properties.

A pair of operations that are related to Sketchpad's concept of instancing are JOIN and SPLIT. By default, when a selection is made, only primitive objects such as lines and points are selectable. To change this, the JOIN function will combine an extended selection of elements into a cluster element. The cluster element will then behave as a primitive element for purposes of editing and selecting. The SPLIT function acts upon a cluster element and reverses the effect of the JOIN function.

Star's integrated text and graphics editor demonstrates an effective and well conceived approach to user interface design. However, the overall usefulness of the editor is reduced by not providing for such things as a curve primitive.

2.3.3. MacDraw

MacDraw is a package produced by Apple Corporation to be used on its line of personal computers [MacDraw84]. Very little new functionality is introduced by this package; however, it does combine a lot of the ideas introduced previously.

MacDraw expands on the ideas espoused by Draw. It too is a grid based system with redefinable grids. It supports multiple brushes and also draws splines. Some ideas from Sketchpad are used as well. These include dynamically changing numbers which represent line length when a line is rubberbanded. Permanent menus reside on the screen, but pop-up menus, which have been used in previous packages such as Star Graphics, are used extensively here. Pop-up or pull-down menus are used effectively to allow the user to select a large number of predefined text fonts, line styles, line thicknesses, in addition to other choices. The user is able to draw freehand sketches but no geometric information is stored. This mode has the disadvantage that each point on the curve must be stored separately, resulting in very inefficient use of memory.

Two selection mechanisms are provided by MacDraw. The first mechanism allows the user to specify individual objects by simply selecting when the tracker is within the extent, or bounding box, of an object. The extent of a line, for instance, is roughly the smallest rectangle, oriented with the coordinate axes, that will fully surround the line. This mechanism can be thought of as a form of 'rectangular gravity'. The second mechanism allows a user to select multiple objects within a rectangular region. Objects are only selected if their extent is fully contained within the selection rectangle. To invoke this mechanism, the user first selects a corner of the rectangle and keeps the selection key depressed. Then, by moving the tracker, a dynamically changing rectangle is displayed. By releasing the selection key, everything within the current rectangle is selected. All system interaction other than text operations, which require a keyboard, is performed with a mouse.

Draw packages previous to MacDraw were mostly research-oriented in that relatively few people other than the designers used them. MacDraw was the first package to gain widespread use among the general public, due to the affordability of the personal computer on which it is used. A very easy to learn and consistent user interface permits novice users to be productive in a short period of time. Basic operations are implemented effectively and are easy to use, but more complicated operations are difficult if not impossible to perform.

2.3.4. Drawing Beautifier

Although the system described in this section only loosely fits into the definition of a draw package, it is nevertheless included because of the way in which it handles constraints. The beautifier is implemented as part of the PED graphics editor [Pavlidis85]. The system works as follows: given a rough drawing as input, desirable constraints are inferred and the drawing is then modified to satisfy these constraints where possible. Constraints or relations dealt with are: approximate equality of the slope or length of sides, collinearity, and vertical and horizontal alignment. A restriction is applied to limit drawings to ones that can be defined in terms of points. This covers most line drawings.

For each type of relation, objects are sorted according to the value of their scalars (i.e., coordinates and line lengths). These lists are searched to find objects that are approximately related. For example, two *x*-coordinates that differ by less than some pre-determined value are flagged. This approach ensures that the search for relations among *N* objects in the drawing can be performed in $O(N \log N)$ time, since the sorting process requires this amount of time. The relations flagged are then expressed as a series of equalities. A problem with this approach is that a series of parallel lines close together could be mapped onto one line. To alleviate this problem, negative constraints are imposed on the system of equations derived above. Negative constraints indicate that entities should *not* satisfy a condition. For example, points that are close in both *x* and *y* coordinates should not be constrained to be equal.

The following constraint relations can be imposed on the primitives of a drawing:

1. Line segments at angles near a preferred value (a multiple of 45 degrees, for instance) are adjusted to the exact value.
2. A set of neighbouring line segments with similar slopes are made collinear.
3. Line segments should have the same lengths.
4. Points are horizontally [vertically] aligned unless they are also vertically [horizontally] aligned.

It is not likely that all the constraint equations can be satisfied simultaneously. To ease this problem, a penalty is assigned to each constraint equal to the maximum change in one of its variables, assuming the other variables remain constant at their original values. This represents a preference to those equations which only require a small change to be satisfied. If a solution to a large subsystem of equations does exist, the solver quickly finds it. However, there is still no guarantee a solution exists.

This system can be thought of as another constraint based draw package in the Sketchpad family. Curves cannot be beautified in the same way as lines, but they can be expressed as spline approximations. The method described in this section was originally designed for use as a filter for edge extraction algorithms on digitized images, since those algorithms produce rough line drawings due to noise in the image. These rough drawings are similar to rough 'sketches' of drawings in an interactive mode.

2.3.5. Juno

A new approach to implementing geometric constraints is presented in the Juno system [Nelson85a][Nelson85b]. Two techniques contribute to the success of Juno: *geometric specification* and *implicit editing*. Geometric specification is the use of geometric constraints to specify locations, such as specifying that points *a, b* and *c* be collinear and that *b* be exactly mid-way between *a* and *c*. The interpreter translates the geometric constraints into a set of numerical constraints on object coordinates. The list of constraints is then solved by iterative methods.

Juno integrates a WYSIWYG image editor with a language like Knuth's METAFONT [Knuth79]. METAFONT was originally a font design tool capable of producing many mathematical symbols and alphabets. However, it is used in other areas of programming with no graphical requirements. Languages of this style seem well suited for producing constrained images with a hierarchical structure. The features which make it suited for these applications are that the positions of points are given with declarative constraints, drawings are produced by drawing commands parameterized by these points, and procedures may be defined.

Implicit editing refers to the automatic modification of the text of a Juno program after the displayed image has been edited interactively by the user. If an object is edited, the program text that draws this object is changed and that section of code is re-executed, updating the screen at the same time. The types of interactive commands that may be performed on an image consist of moving points, adding or deleting points, specifying constraints, and painting commands.

Previous constraint-based systems dealt with linear constraints, such as horizontal alignment. Linear constraints are those in which the algebraic relationships among the constrained components can be expressed as a system of linear equations. This excludes quadratic constraints such as those found in a parallelism test. Juno will handle both linear and nonlinear constraints.

Two dangers arise from allowing this improvement: the constraint solver may become too slow or its solutions might become unpredictable. Concerning the first danger, Juno uses Newton-Raphson iteration [Burden81] in its constraint solver. This derivative-based method is reputed to be faster than the relaxation methods used in Sketchpad.

Experience with this method in Juno has shown that its performance is predictable and reasonably fast. The second danger is of no concern if the solution is not too far from the initial configuration, as nonlinear solvers behave predictably for these cases. So, when using the editor, the user must lay out the points in roughly the correct positions, and the constraint solver will align them accurately. Similarly, a Juno program must provide hints for unknown points when the constraint solver is invoked.

The commands available in Juno can be thought of as operations to change the state of an "abstract Juno machine". The state of the machine is determined by:

- a collection of *point registers*, each containing a value of a point in the Cartesian plane.
- a single *image*, physically represented as a grid of pixels.
- three *mode registers*: colour, which controls the drawing colour; width, which controls the line width; and endsType, which controls the drawing of ends of lines.

The colour register has a limited number of values that it can be set to. It contains the current drawing colour. The width register is set to a decimal value that indicates the current line thickness, while the endsType register can be set to three different values: BUTT, SQUARE or ROUND, depending on the desired type of end wanted on the current line. The commands to set these registers are as follows:

•`c PAINT A` :	set the colour register to `c`, execute the command `A`, then restore the contents of the colour register.
•`e ENDS A` :	set the endsType register to `e`, execute `A`, then restore the endsType register.
•`r WIDTH A` :	set the width register to `r` millimetres, execute `A`, then restore the width register.
•`p,q WIDTH A` :	set the width register to the distance between points `p` and `q`, execute `A`, then restore the width register.

The list of painting commands, or commands which actually create images, is as follows:

•`FILL p` :	change the colour of every point enclosed by path `p` to the current value of the colour register.
•`STROKE p` :	draw a stroke along path `p` of the current colour, width, and endsType.
•`DRAW p` :	draw a stroke along path p that is black, has round ends, and has width 1.

In the above commands, a path is a connected sequence of edges and arcs, formed by either endpoints of lines or control points for Bezier splines.

Only four constraint types are allowed in Juno, these being:

- `(x,y) CONG (u,v)` : the distance from x to y equals the distance from u to v.

- `(x,y) PARA (u,v)` : the direction from x to y is parallel to the direction from u to v.
- `HOR (u,v)` : the direction from u to v is horizontal
- `VER (u,v)` : the direction from u to v is vertical

This small group was chosen because any constraint can be expressed in terms of `CONG` and `PARA`. `HOR` and `VER` are included to orient images on output devices.

For some people, the meanings of the `HOR` and `VER` constraint types may be non-intuitive. The meaning of `VER(u,v)` should be taken as 'the point u is at the same *height* above the *x*-axis as point v'. This is the standard used in many typesetting systems. It should not be confused with the notion of u being vertically above v, which is expressed by `HOR(u,v)`. A user must therefore be careful when defining operations using these constraint types.

Juno's constraint command takes the form:

LET Variables | Constraints `IN` Command `END`,

where Variables is a list of variables to be used locally within a procedure, Constraints is a conjunction of the constraint types applied to the variables, and Command is a list of painting operations. The Variables list may contain hints in the form of condition(s) for specific variables within the list.

Following is an example of how these constraints and commands can be combined to produce a constrained drawing. The problem is to construct an equilateral triangle on the line segment ab. The program would look as follows:

```
LET e == (1,1) REL (a,b) |
     (a,e) CONG (a,b) AND (b,e) CONG (a,b)
   IN DRAW (a,b),(b,e),(e,a)
END
```

Here, e is introduced as a local variable and represents the third vertex of the triangle. The first line of the program is giving the constraint solver a hint for the value of e. Assuming a is the origin of the coordinate system and b is at the point `(1,0)`, the initial value for e is at `(1,1)`. This prevents the construction of an unpredictable triangle, or an equilateral triangle with the third vertex below the segment ab. The second line applies the constraint that the two lines to be constructed should be the same length as ab. Finally, the third line contains the graphic commands which construct an image from the solution of the constraint solver.

The user views two windows on the display screen; one displays the unparsed text for all the currently available procedures, and the other displays the image produced by the current command. The text window is essentially a WYSIWYG text editor. After editing the program text, the user enters the 'Reparse' command, which will regenerate the list of procedures used, and then regenerate the image. The image window can be edited graphically, as in other draw packages. The difference with Juno is that this editing will result in modifications to the text of the current command.

Juno has an iconic menu from which to invoke operations. The operations available consist of: drawing straight lines and arcs; entering text; moving, copying, and erasing; freezing points; imposing parallel, equidistant, horizontal and vertical constraints; and creating and calling procedures. Freezing a point causes its location on the screen to remain constant during all operations until it is unfrozen. To invoke an operation, the user picks the appropriate icon, specifies the argument list for the command and then hits the escape key on the keyboard. The argument list is a number of points which are selected by pointing and clicking a mouse button. Another way to select a group of points is to draw a free-hand circle around them. It is then possible to perform a transformation on the group as a unit.

Juno's approach to a constraint-based system is considerably different from other implementations. The procedural abstraction is useful, but using a programming language style automatically restricts the number of people who use this system. While allowing commands to become parameterized procedures, Juno does not allow a group of constraints to become a parameterized predicate. For example, it would be useful to be able to express a right angle formed by `a`, `b` and `c` by `Right(a,b,c)`, instead of including a list of primitive constraints every time.

Two pitfalls in expressing constraints are to be avoided when using Juno. First, the constraint `(a,b) PARA (a,c)` is more stable than `(a,b) PARA (b,c)` if `b` and `c` are near one another, and `a` is far away; the solver will tend to collapse a short line segment to a point if it is parallel to a long segment. Secondly, redundant constraints will sometimes prevent the solver from converging to a solution.

2.3.6. ThingLab

Yet another approach to defining constraints is taken in ThingLab [Borning86]. ThingLab is a constraint-oriented simulation laboratory which can be used for constructing dynamic models of simulation experiments such as those mentioned in Sketchpad. The unique aspect of this system is that the user defines constraints graphically. The reasoning behind this approach is that less experienced users will find it easier to add new constraints than they would using systems such as Juno, where constraints are specified in an underlying language outside the graphical domain. Two sorts of users use ThingLab; one constructs a set of building blocks and one uses the building blocks to perform particular simulations.

The original version of ThingLab required that constraints be defined by writing pieces of Smalltalk code. The user had to define a predicate for the constraint, as well as one or more methods to be used to alter parts of the object to make the constraint hold. For example, in order for a point to be constrained to lie exactly at the midpoint of a line, the predicate would be the equation:

$$\textit{midpoint} = \frac{\textit{point}\,1 + \textit{point}\,2}{2}$$

and a method used to satisfy this constraint would be to simply calculate the right hand side of the equation and assign the result to the midpoint. In the current version of ThingLab, only the constraint's predicate need be given; the system will derive the methods to be used.

To create the building blocks, ThingLab contains an *object definer* window. Besides a few menus of commands, this window has two panes: a *definition pane* and an *instance pane*. The definition pane is where constrained objects are defined. This window contains the graphical definition of the constraint along with a list of the object's parts and inserters. Inserters are parts of an object that are to be positioned when a copy of it is inserted into a construction (e.g., endpoints are the inserters for lines). The instance pane simply contains a depiction of what will be added to a construction if this object is instanced. When the user adds an object to some other object, the instance pane is copied; the definition pane is not copied, but the instance will behave in accordance to the constraints in its definition.

The graphical definition of a constraint is in the form of a network. The network shown in Figure 2.2 is used to specify the constraint that a point M should lie mid-way between points P1 and P2:

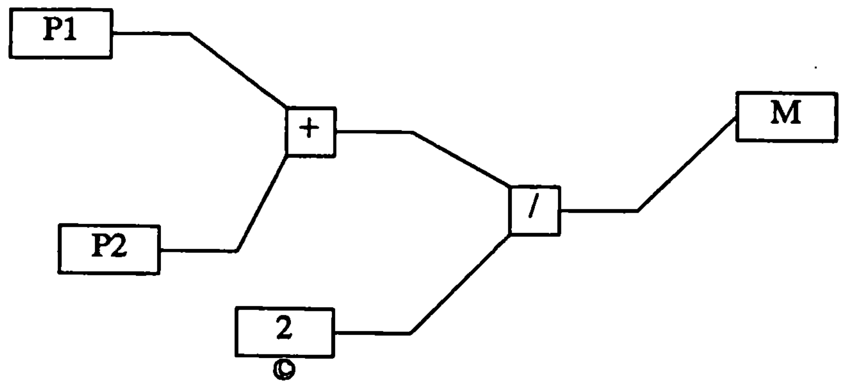

Figure 2.2 — A constraint network

The symbol under the box containing '2' indicates that this element should be kept constant. The other boxed variables (P1,P2,M) are free to change. Other possibilities for constraints include vertical lines and forcing digits of text to correspond with the size of an object.

For simple constraint types, the networks are reasonably easy to understand and construct. However, complex constraint networks can take up considerable space and can be difficult to understand. Although ThingLab introduces an interesting method for specifying constraints, it is debatable whether it is more efficiently done graphically or textually, as is done in Juno.

2.3.7. Gargoyle

Gargoyle [Bier86] uses an interaction technique called *snap-dragging* in a two-dimensional illustrator. The idea behind this technique is that a set of gravity-active *alignment objects* are automatically constructed during the editing process to aid in construction. Heuristics about typical editing behaviour plus hints from the user determine when and where these objects are created. Snap-dragging may be thought of as a simple constraint solver or as an extension of the gravity-active grid concept. Points can be placed to satisfy constraints such as angle and distance, but constraints are forgotten as soon as they are used.

To position objects with precision, a special gravity-active point called the *caret* is used. The caret moves with the cursor, but can stray away if attracted by a gravity-active object. The cursor is not attracted by gravity. Control points are always added to a scene at the position of the caret.

The user selects what type of alignment objects are to be active, and in which region they are to have an effect. These objects may be points, lines or circles, and are coloured grey to distinguish them from the rest of the scene. Whenever a point or an object is moved within the region, the alignment objects become visible and attract the caret, aiding positioning. There are several ways provided to disable the alignment objects when the user so desires.

The usual types of transformations such as scaling, translation and rotation may be applied to objects. To do so, one first selects an object, then places the caret at an initial point in the scene. The caret is then moved while the objects are transformed based on the displacement between the initial caret position and the current one. To help in rotation and scaling, an anchor is placed on a point to prevent that point from moving during the transformation. As an example, in a scaling operation, the scale factor is the ratio of the magnitudes of the current caret displacement (caret – anchor) to the initial caret displacement.

Snap-dragging only provides commands that take one or two arguments. The decision not to allow more arguments is based on ease of use of the system. Although more arguments would increase the power of Gargoyle, it would also result in a less friendly user interface. The two-argument limit was thought to be an appropriate balance between these conflicting goals. Another aspect of the user interface is an undo operation, which is provided in Gargoyle as well as incremental checkpoints of the system state to prevent catastrophic failures that would result in the loss of complete or partial drawings.

To enable the construction and placement of alignment objects, two steps are taken. First, the user must specify the set of points and segments at which the objects are to be constructed. These points and segments are then deemed to be *hot*. The second step is to specify the types of alignment objects wanted. The types of objects available are lines of various slopes, circles of various radii, and points of various displacements. The values for these varying scalar quantities are selected from menus. The menu for the slope of a line may contain values that are multiples of thirty degrees. The user may add or delete menu items when he wishes. An extremely useful facility is the ability to measure quantities from objects on the screen. The quantities that can be measured are: the distance between a pair of points, the slope between a pair of points, the angle between three points, and the distance from a point to a line. The measurement is accomplished by placing the caret successively on the points of interest and invoking the 'measure' command. The result of the measurement can then by placed in an appropriate menu.

The hot vertices and segments are combined with more vertices and segments that have been chosen by heuristics. An example of a heuristic used in Gargoyle is that when the user moves one point of a polygon, he will often want to align it with other parts of the same polygon. In this case, the set of vertices and segments of the polygon are added to the hot list. If any members of this new list are being moved, they are removed from the list; all alignment objects are stationary. Finally, the set of constructions specified is consulted, and each point and segment in the final list is made the target of a construction. For example, if alignment lines of thirty degrees were specified as the alignment objects, a thirty degree line would be drawn through each vertex in the final list.

Snap-dragging uses the ruler and compass metaphor to create an environment that is much like that in which a draftsman operates. While grids provide only a small amount of drawing aid, and constraints provide a great deal of power but often at the expense of understanding and manipulation, snap-dragging provides much of the power of most constraint-based systems without the need for additional understanding of any underlying structures. An obvious drawback is that setting up the alignment objects may take a long time. As well, the construction of alignment objects and calculation of many intersection points is computationally expensive.

2.3.8. Other Interactive Packages

In this section, a number of other packages are briefly surveyed, and their importance explained. Most of the ideas here deal with different user interface techniques.

William Newman's 'Markup' [Newman75] is a tool for interactively editing mostly textual documents. This is really more of a paint program since it provides pixel editing, but it does provide rudimentary drawing operations in a grid-based system. When introduced, Markup was especially noted for its ability to combine text and graphics in a single document. The special features of Markup are the early use of pop-up menus and the way adjoining lines are treated. If two lines join at an acute angle, for instance, an 'arrow' will be formed at the intersection instead of a messy join.

Gremlin [Opperman84] is a useful tool for creating typeset drawings in TROFF [Ossanna79] format. Drawings are created by Gremlin's image editor and can be output in a virtual file format. If the outputted file was named 'test', then by placing the following lines in a troff format file, the drawing will appear in a 3x5 inch box in the text of the document, after the troff file has been run through the Gremlin filter:

```
.GS
  file test
  width 3i
  height 5i
.GE
```

Gremlin is a relatively simple draw package with tablet or keyboard input. It is a grid-based system that has four different font types and variable thickness lines of any size that troff will support. Only one text font is actually drawn in the image editor and lines of different thicknesses are drawn at the standard thickness but in different colours. However, the size and font information is present in the virtual output files.

There are a few peculiarities unique to Gremlin. Four image buffers are available. An entire image on the screen or any selectable subpart can be copied to a buffer and later copied from the buffer to the image. One of the tablet buttons is able to sequentially undo approximately ten previous steps. This seems excessive and can lead to unpredictable results. There is no way to define an object hierarchy; a connected series of primitive objects cannot be selected as a unit by pointing to an individual primitive. However, the group can be selected by an area selection.

Finally, Gremlin reverses the practice of selecting an operation and then selecting its arguments. Here, the user selects a number of arguments first, which are labelled by a sequence of numbers. Next, the operation to be performed on these arguments is selected and the construct is drawn. This is referred to as *postfix* notation. Other packages use the *prefix* notation for commands in which the command is selected first and its arguments are selected next. In such packages it may be difficult to describe a complex selection such as 'all objects except the following objects' since multiple selections need to be made and the interpreter needs to be able to find the end of the command. In packages such as Juno, the end of the command needs to be indicated by pressing the 'escape' key on the alphanumeric keyboard.

Often a user of an interactive system has a limited number of input devices at his disposal. In response to this, a number of conceptual models of devices have been developed [Wein81][Evans81]. From these models, it is easy to control operations like rotating and scaling with almost any type of tracking device. A *slider* is a method of inputting a scalar value by positioning a pointer at a location on a stationary scale [Buxton82]. A *turntable* [Wein81] is a model used to control rotation about an arbitrary point. Conceptually, one can visualise this operation by controlling a turntable placed directly underneath the rotated object. Echoes are drawn for the centre of rotation and for the current rotation applied. The tracker controls the movement of the turntable; the closer it is to the centre of rotation, the greater the effect a movement will have on the rotation.

A model related to control scaling about a centre of scale is called a *rack* [Wein81]. A rack is attached underneath the scaled object so that the object may be condensed or stretched by control of the tracker. Two echoes besides the tracking icon are seen by the user. One is stationary and indicates the centre of scaling. The other is controlled by the tracker and indicates the amount of scale applied. By moving the scaling echo, the object is dynamically scaled. The user has the ability to specify if the scaling is to be done in the x-axis, the y-axis, or in an equal amount in both axes. A *stirrer* [Wein81] is a model that can be used in both scaling and rotating operations. In a rotation operation, the centre of rotation is selected as with the turntable, and the same echoes are provided. To affect a rotation, the user 'stirs' his input device. With a tablet, stirring is accomplished by rotating the puck or stylus in a clockwise or counter-clockwise direction. The faster one stirs, the greater the rotation applied to the object.

A number of user interface considerations were applied in the development of both Griffin [Stone80a][Stone80b] and a prototype picture creation system [Plebon82a]. These systems have a similar hierarchical structure in the object database. That is, instead of allowing a drawing of a rose, for instance, to be selectable only as an entire rose or by its individual primitives (i.e., lines, curves, etc.), the user may create a hierarchy whereby he may select the entire rose, individual petals or thorns, the stem, or individual primitives. This hierarchy may be traversed; one method is to assign buttons on a puck the functions of either moving up or moving down the hierarchy. By pressing a button, one can go from selecting the stem to selecting the entire rose. By pressing another button, a component line of the stem can be selected. The system changes the default selectivity level in response to user actions. For example, if the user was just moving a petal, subsequent selections would select objects at this level in the hierarchy, whereas if he were to then start entering lines, individual lines or curves of the petals would be selectable. The user may also set a default level of selectability in the hierarchy.

Attention was also paid to the user interface presented by menus. Menus in these systems are not required to stay at the same location on the screen. The user may reposition them as he desires, and can save these positions for future editing sessions. Information is kept about the traversal through the system of submenus. Thus, if a user wants to return to a main menu, he does not have to explicitly retrace his steps.

2.3.9. Non-interactive Packages

There are many interactive packages available, but often one does not have the necessary hardware such as a monitor or adequate input devices. For these cases some form of textual language is used to specify the structure of a drawing. Commands to generate the graphic primitives for the drawing are output and can be used to create a visual image. Usually one is able to include these images within the text of a typeset document. Two of the most popular language-driven packages in use today will now be introduced.

PIC [Kernighan81] is a language for drawing simple figures on a typesetter. Capabilities exist to draw primitives such as boxes, circles, ellipses, lines, arrows, arcs, spline curves and text. Once the text file containing the PIC commands is created it may be passed through a filter to generate a TROFF format file. In this way it is easy to integrate text and graphics in a single document. The language is very low-level, as no control structure or procedure mechanism is provided. Objects may be placed at an absolute position or relative to the current drawing point. Coordinates are usually in units such as inches or centimetres to correspond with the size of the hard copy image. All graphic primitives have a default size which can be changed temporarily or permanently. Text may be included inside the drawing, but PIC does not keep much information about the size of text relative to the size of objects. Other features of PIC are:

- the spacing between dots and dashes may be controlled in dotted and dashed lines
- objects may be made invisible to aid in tasks such as positioning
- the attributes of an object (i.e., size, line type, etc.) may be copied to the subsequently drawn object
- objects may be labelled and referenced later
- most objects have labels applied to their most significant points (e.g., a rectangle will have addressable locations at its compass points (i.e., north, south-west, etc.))
- text may be left, right or centre justified
- if a line intersects a polygon edge, it may either be clipped at the edge or drawn through
- primitive objects may be grouped together to act as one object
- the state of the system (i.e., current drawing position, direction of motion, etc.) can be left unaltered by enclosing a group of commands within brace brackets
- macros may be defined.

PIC requires a great deal of time to lay out a drawing before any commands are written. Furthermore, one does not know if the drawing generated is correct until some visual representation of it is viewed. A feature that is lacking in PIC that is available in other packages is the ability to perform transformations on objects. To rotate a rectangle, one must apply the transformation by hand and also convert the definition of the rectangle to four lines.

IDEAL [VanWyk82] is a high level graphics language using a procedure mechanism to create two dimensional images that can be included in TROFF source files. Picture elements are referred to as *boxes* in IDEAL. A box is defined by declaring any algebraic relationships that should hold among points and requesting actions that should be performed at those points. Here, the user expresses the relationships that should hold among variables as opposed to a sequence of assignment statements that accomplish these relationships, as is done in packages like PIC.

Boxes act as procedures in IDEAL. Each box has a number of internal variables which may be used as parameters to define a box. In this way, a drawing element may be generated by specifying values for any subset of the box's internal variables that completely defines the object. An equation solver is used to process the algebraic relationships and generate the absolute positions of significant points. For example, a circle may be created by specifying its centre and radius, three points on the circle, or even two points on the circle and one coordinate of the centre. IDEAL draws pictures on the complex plane and hence it supports the usual complex arithmetic operators. Scalar values are treated as a vector with a null imaginary component.

Operations provided by IDEAL are:

- any type of figure that can be defined as a box can be used as a pattern for a line
- regions may be filled with patterns
- any area within a boundary can be made 'opaque', to hide any portion of the picture behind it
- transformations can easily be applied using complex arithmetic.

While PIC may have a greater number of graphic primitives than IDEAL, it is less flexible and less powerful. A drawback of IDEAL is its use of complex numbers. It is unreasonable to assume that all users know enough about complex arithmetic to create elaborate drawings.

2.4. Trends in Draw Packages

While many advances have been made with draw systems, there are still areas in which improvements can be made. Developments in draw programs tend to occur infrequently and usually involve significant differences from existing programs. The areas in which current developments are taking place are the user interface, hardware, and management of data.

Much has been mentioned about user interface techniques in the previous section. The majority of cases have had to do with the manipulation of objects in a scene or ways to invoke functions. A more recent development is the use of natural language interfaces in draw and CAD systems [Vernadat84]. In such an interface, interaction would not be performed by a sequence of selections from an input device, but rather with phrases from a language such as English. For example, a user may be able to enter a command such as 'Change All Solid Red Lines To Dashed Blue Lines', instead of having to select an operation and its arguments with an input device such as a mouse.

Further improvements may be achieved by more effective input devices. Such devices may give a more natural 'feel' for the manipulation of screen objects and may improve factors such as the speed of interaction.

Finally, a great deal of research has be done in the field of User Interface Management Systems (UIMS) [Thomas83]. A good UIMS is a programming tool that, because it allows one to create user interfaces quickly and to modify them several times during the design process, encourages the designer to iteratively approach a better interface. However, any interface produced by a UIMS could probably be improved upon by hand coding. Most UIMS's developed to date limit the ability of an end user to take advantage of parallel forms of input from multiple input devices.

Developments in hardware make some computationally expensive operations possible that were previously infeasible due to feedback constraints. Such developments include more powerful microprocessors, multiprocessor systems, and special purpose graphics hardware. Advances in CRT technology, in particular screen resolution, create a more natural drawing surface that makes artifacts such as aliasing and screen flicker less noticeable.

Recently, much attention has been directed towards Data Base Management Systems (DBMS) [Vernadat84], particularly with larger CAD systems. The application data structure of the image could be maintained in a database. Queries to the database are then easily constructed in the application. A DBMS relieves much of the tedium that the application program must otherwise do to manipulate storage and provide capabilities to add, delete and access data. Other advantages of a DBMS include:

- complex relationships between data can be represented
- data redundancy is controlled
- data integrity and consistency is maintained
- data security and recovery are possible

In the area of data models, improvements are being made in the logical arrangement of picture entities. The greatest developments are being made in CAD packages and in 3D draw packages. Developments in two dimensions are confined to representing hierarchies and creating structures in which either common or expensive operations can be done efficiently.

Proposed Implementation of a 2D Draw Package

3.1. Introduction

This chapter describes the proposed and partially completed implementation of an interactive two dimensional draw package capable of producing high quality images in a format suitable for display on many output devices. Eventual output would be in a virtual file format. Various filters could then be used to generate a representation of different formats, including formats for typesetting. High quality images have few observable spatial or chromatic artifacts.

This particular implementation is designed for use on the Adagio stand-alone workstation [Tanner85], although it could easily be used in a timesharing environment. A single user workstation is more appropriate for interactive applications because feedback is time critical. The design of this implementation will closely follow the ideas and drawing methods present in Gargoyle.

The original motivation behind this implementation was to create high quality images for use as stimuli in psychology experiments being conducted at the University of Waterloo. For these purposes, only two dimensional drawings are necessary. The Gargoyle design was chosen as a model due to the author's biases. Of all the packages listed in the previous chapter, none were implemented using a multitasking approach.

The first section of this chapter describes the existing environment of the implementation in terms of hardware and software, and a user's view of the environment. Next, the design of data structures such as the display list and the application data structure are described. The task structure of the system with emphasis on communications between tasks is described afterwards. Finally, suggested improvements to the system are listed and the state of development of the system is described.

3.2. Environment

3.2.1. Hardware

The Adagio workstation is based on the Ikonas RDS-3000 frame buffer system [Ikonas82]. It consists of a 1024x1024x24-bit image memory, a frame buffer controller, a crossbar switch, colour lookup tables and D/A converters. Attached to the Ikonas bus are a custom bitslice processor, a video input card, scratchpad memory, a matrix multiplier, a host interface to a VAX 8600 computer, and a Multi Peripheral Controller (the MPC). The MPC is a single board Motorolla 68000-based processor with a Peripheral Control Processor (the PCP).

The particular Ikonas used for development does not contain a character generator even though it can be configured to handle one. Such a device would permit the fast display of text of various sizes and fonts. Since the image needs to be drawn quickly, this implementation does not support text within the image being created. Any text that is needed on the screen is written pixel by pixel by an application routine.

The bitslice, or bipolar processor (BPS), is a highly specialised processor which writes the working image into the image memory. Whenever a new image is to be created, the bitslice accesses the display list and draws each primitive into memory. The matrix multiplier provides hardware assistance to the bitslice, such as point transformations and multiplying transformation matrices.

The PCP contains parallel and serial ports plus A/D converters which can support a number of I/O devices on the MPC. Software which resides on the MPC controls the flow of this input data. The application software which manipulates the data structure of the drawing also resides on the MPC. Both the MPC and the bitslice processor are able to set the hardware video registers on the Ikonas which permit control parameters such as window locations, window size, etc..

3.2.2. Software

Adagio uses the Harmony operating system [Gentleman85], a multitasking, multiprocessor, message passing system, although only one processor is used in this workstation, that being the MPC [Forsey85][Booth85]. Harmony is designed for real-time systems that use multitasking for the simplicity of design and multiprocessing to achieve performance. All software development is performed on a VAX under Unix. Compiled programs are then downloaded onto the MPC and executed.

Communication and synchronization among Harmony tasks is achieved through four message passing primitives: `_Send()`, `_Receive()`, `_Reply()`, and `_Try_Receive()`. A complete description of these primitives and the rationale for choosing them is provided in [Gentleman81]. Tasks can be created and destroyed dynamically using: `_Create()`, `_Destroy()`, and `_Suicide()`. Communications with external devices uses `_Await_Interrupt()`.

The ability to create and destroy tasks dynamically provides one with simple structuring techniques. This is analogous to hiring a temporary employee to perform a specific job and firing him immediately afterwards. The process of designing software in this fashion is called 'anthropomorphic programming' [Booth82].

Central to the structuring of workstation applications is the idea of a *resource abstraction*, in which one or more tasks provide a service to other tasks. *Client* tasks request services from *server* tasks. Often a server is used to control the mutually exclusive access to a scarce resource such as a data structure or output device. It thus ensures that the resource is kept in a consistent state. Another possible configuration that can be used instead of a server is an *administrator* with a group of *worker* tasks. The administrator schedules and manages the workers who perform the requested services. Other important task types are *couriers* and *notifiers*. Couriers pass messages between two tasks and are usually used whenever the sending task does not want to wait, or become *blocked*. The sending task sends a message to a courier which then sends the message to the receiving task. The courier waits for the receiving task to respond, while the sending task is unblocked. A notifier is often used in cases where a task needs to wait for an event but does not want to be prevented from doing other things in the mean time. The task creates a notifier and can then continue with what it was doing. Meanwhile, the notifier waits for the event and then informs the parent task.

Programs for the BPS are executed in microcode. Source routines are written in Microcode C, which supports a subset of the C programming language. A Microcode C cross compiler generates the microcode for the BPS [Gurd83][Gurd85].

The implementation described in this chapter uses ARIA [Loo86], a near-real-time graphics package designed for the Ikonas, to draw the images. ARIA supports the rapid updating of a display of geometric objects, including the dynamic updating of transformations. A set of software routines is provided which gives the user the ability to manipulate the display list. A library of these routines can be linked together with application software to reside on the MPC. ARIA also provides the BPS microcode which traverses the display list and writes to the frame buffer memory.

ARIA divides the image memory into four quadrants of 512x512x24-bits. Two quadrants are used to double buffer the image displayed. As one quadrant is displayed, the display processor can write into the second. When the image in the second quadrant is fully drawn, the visible portion of image memory is changed so the second quadrant is displayed. The first quadrant can then be redrawn. By doing this, the user does not see the flicker on the screen that would be caused by continuously erasing and redrawing. A third quadrant is used for a z-buffer, which is not used in this application, and the fourth quadrant is unused.

As mentioned previously, the application software which manages the data structure of the image runs on the MPC. This data structure has to be constantly updated in response to user input. The structure maintained contains hierarchical picture information and will be in a format completely different than that of the display list. The purpose of storing a separate data structure is to allow abstractions of groups of graphic primitives. The display list is stored as a directed acyclic graph (DAG) and only knows about primitives such as points, lines, etc. With an application data structure, one may manipulate higher level objects. However, the application must be able to convert the data structure into a display list quickly.

When a user modifies the image on the screen, the application changes its data structure. It must then modify the display list to reflect these changes. The BPS must then be informed that the image should be redrawn. All these steps are achieved by ARIA procedure calls.

The display list is kept in a separate area of the Ikonas address space called scratchpad memory (SPAD). Here, the MPC is able to write the display list and the BPS is able to read it. To synchronize the MPC and BPS, a word of scratchpad memory is used as a flag. The BPS continually reads this word until it becomes non-zero. When it does, the value in the word is the starting address of the display list in SPAD memory. When the BPS is finished traversing the display list, it sets this word back to zero. Thus, whenever the MPC wants the image to be redrawn, it simply writes the starting address of the display list into the flag word in scratchpad memory.

3.2.3. User

When a user sits down to use this system, he will see in front of him: a high-resolution video monitor, a terminal with a QWERTY keyboard connected to a VAX, a tablet with either a puck or stylus, and a collection of other input devices. Images created by the user are viewed on the video monitor. The terminal is used for system messages and for browsing through the Unix file system to retrieve or save an image in a virtual file format. The keyboard is used whenever an alphanumeric entity is input, such as file names, specific lengths of lines, text to be included in the image, etc. The majority of input actions are performed with the tablet, although other devices may be more appropriate for some tasks, such as a knob for rotation. However, this system is designed so that an operation can be performed with many input devices, and many input devices may be used on different operations at the same time.

In general, operations to be performed are selected from menus. Operations such as selecting parts of the image are 'modeless', in that they do not have to be selected from a menu, and in this case, are the 'default' operations. Other operations may involve manipulating figures such as sliders to change a drawing parameter such as the thickness of lines. The video monitor screen will contain fixed, constantly displayed menus, areas for data entry such as slider windows, and one or more windows to display the image being drawn. Pop-up menus are also available for some operations.

The screen space that is not used for menus and sliders is available for windows of the image. In the two dimensional case, different windows may either represent a detailed view of a portion of the image or a view with various parameters changed, such as line colours, thicknesses, etc. A user may interact with more than one window at a time.

This implementation follows the ruler and compass style of drawing introduced by Gargoyle. Thus, creating, manipulating and erasing alignment objects needs to be addressed. A separate section of the space reserved for menus has to be allocated to handle the operations on these objects. Alignment objects will appear with the image being drawn, but will always be coloured grey.

The following definitions will be needed for the remainder of this essay:

- A *primitive* is one of the most basic objects that can be drawn. For this implementation, a primitive can be a point, line, polygon, circle, ellipse, or spline. A polygon is stored and manipulated differently than a closed sequence of lines.
- An *object* is defined to be a collection of primitives that are stored together as one entity.
- An *instance* is a representation which does not replicate all the data of an object for each image. Modifications to an instanced object will result in changes to every instance of that object. This is contrasted with a *copy*, which is a separate entity, and duplicates all the storage of the object.
- A *cluster* is a number of objects which have been grouped together to act as one entity.
- An *attribute* is any characteristic of a primitive besides vertices or control points. Examples are visibility, stroke width, aliasing versus anti-aliasing, colour, etc.

The operations that will be available to a user consist of the following:

1) Select a vertex, primitive, object, cluster, or area.
2) Create, delete and manipulate windows.
3) Draw a primitive.
4) Delete a selection.
5) Copy a selection to another location.
6) Make another instance of a selection.
7) Transform a selection.
8) Save an image on a Unix system, and retrieve an image from a Unix system.
9) Cluster a groups of objects and uncluster a cluster.
10) Copy properties from a primitive, object or cluster to another primitive, object or cluster.

11) Change drawing parameters such as colour, stroke thickness, stroke type, polygon filling or wire-framing, aliased or anti-aliased drawing, etc.
12) Control distances and angles with alignment objects.
13) Save and restore a session.

Operation 1 is a modeless operation. Creating a new window is achieved through menu selection while all other window operations use a form of pop-up menu. Operations 3 through 9 and 13 are selected from the fixed menus. Operation 10 uses a combination of fixed menu items and a pop-up menu. When drawing new primitives, the attributes are identical to the settings defined in the current drawing mode. These attributes are either entered with sliders, or are selected as one of two discrete values. The operations described in operation 11 are achieved through the use of the same pop-up menu used in operation 10. Operation 12 uses its own menus and was discussed earlier. All these operations will be described in detail in a later section.

3.3. Display List

The display list used in this implementation is that described by ARIA with a few modifications. It is in the form of a DAG and is manipulated by software routines on the MPC. DAGs naturally support the multiple instancing of objects as well as multiple viewports.

A DAG consists of nodes connected by two types of edges: *right* edges and *down* edges. If a down edge joins node x with node y, x is referred to as the *parent* of y, and y as the *child* of x. Similarly, if a right edge joins node x to node y, x is the *left sibling* of y and y is the *right sibling* of x. A node in a DAG may have at most one down edge and one right edge. When the BPS traverses the DAG, it does so in a depth first order.

There are seven types of nodes used in ARIA. These nodes are *control, instance, transformation, attribute, viewport, display,* and *perspective*. The DAG must be rooted with a control node. Instance nodes point to areas of memory which contain objects, and are always leaves on the DAG. Transformation nodes contain a 4x4 transformation matrix which is applied to every instance node beneath it. Attribute nodes enable the user to temporarily change the value of attributes during the traversal of the DAG. Viewport, display and perspective nodes control the appearance of viewports on the screen, and their view of the image. Each node is identified by a unique positive integer.

Objects consist of a list of vertices and a list of primitives. The primitives have pointers into the vertex list. Each list is arranged so that the last entered element is placed at the bottom of the list. This provides an undo facility when adding primitives. Objects can only be modified if they are *open*, and only one object can be opened at a time. Objects, primitives and vertices are also identified by three different sets of unique positive integers.

Each primitive has associated with it a number of attributes. These include colour, visibility, highlighting, anti-aliasing, stroke width and stroke style. Routines exist that allow the settings of these values to be changed at any time. A distinction must be made at this point between attributes and flags. Attributes have already been defined, but flags are a new concept. The *flags* of a primitive are a subset of its attributes. Flags consist of every attribute of a primitive except colour, anti-aliasing, stroke width and stroke type. To specify values for a primitive's flags and colour, one must first enter the values into global tables managed by ARIA. Two tables are currently in use; a colour table and a flags table. Once the values have been entered into the tables, they are subsequently referenced by their index into the table. Attributes other than colour and flag entries are stored along with the primitive definition (i.e., without an extra level of indirection).

The changes that are needed in ARIA for this implementation are: the ability to draw more primitives, such as splines, circles and ellipses; the ability to draw strokes of variable width and of different brush styles; and the ability to draw anti-aliased strokes. Not only does this require new microcode for the BPS, but new data structures are also needed for new primitives.

3.4. Application Data Structures

This section describes the data structures that the application uses to keep track of the display list and the image on the screen. The data is divided into three sections: information concerning the window layout, a structure which describes the alignment objects, and of course a structure which describes the constructed image. Each structure is in the form of a linked list and is manipulated by application routines.

The window data structure is the simplest of the three. It is a doubly linked list of nodes, where a node contains all the information concerning a window. This information consists of a window id, all the parameters of the viewport, perspective, display, transformation and attribute nodes, as well as their ARIA node ids. If alignment objects are displayed in this window, a flag in the window node is set. If the window is not currently displayed (i.e., it is not connected to the DAG), another flag is set and stored in the node for this window. This state is completely different from deleting a window. Upon deletion, the node in the data structure for this window is unlinked from the list and its memory is freed. The ARIA nodes for this window are also unlinked from the DAG and deleted. When the flag controlling the display of a window is set, the window is temporarily removed from the display but can easily be redisplayed.

The data structure representing the alignment objects is a little more involved. An example of this structure is depicted in Figure 3.1. At the head of the structure is a node which stores the parameters of the attribute node for all the alignment objects, as well as its ARIA node id. Below this node is a doubly linked list of cluster nodes, one cluster for each type of alignment object. The information stored in each node consists of node ids for the attribute, transformation and instance nodes, the object id, the cluster id, the type of alignment objects, and the parameters of the attribute and transformation nodes.

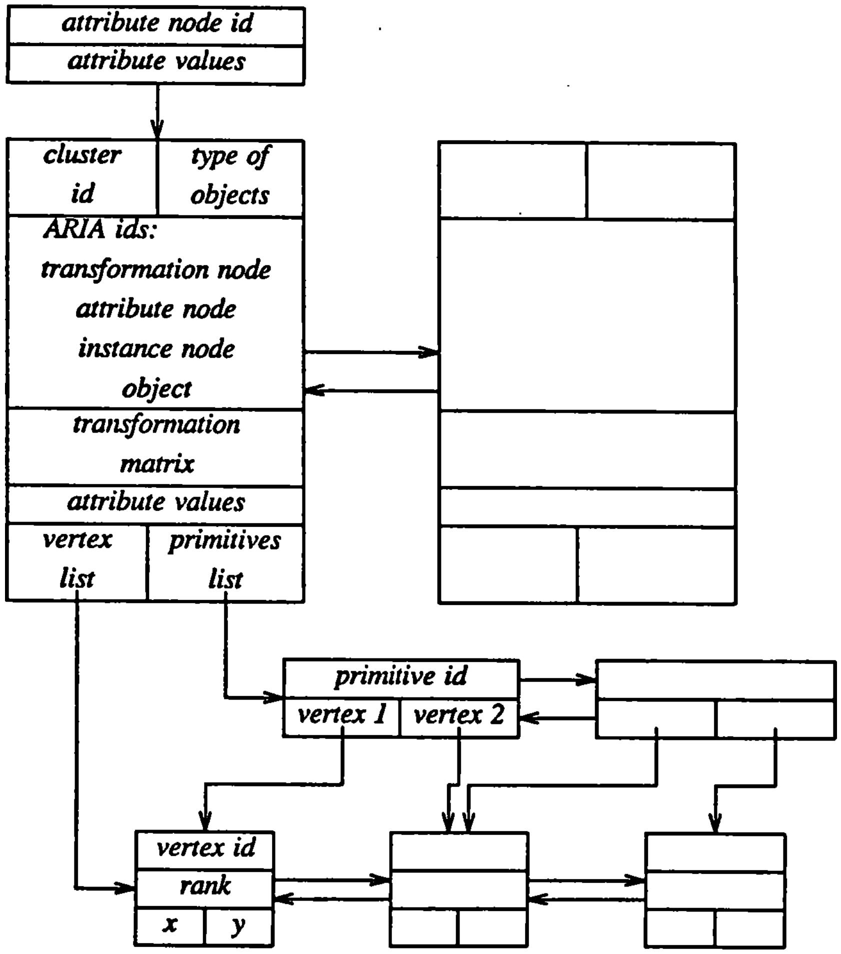

Figure 3.1 – Data structure representing alignment objects

Linked below each cluster node are doubly linked primitive and vertex lists. Each node in the primitive list represents one alignment object and maintains the primitive id and pointers into the vertex list. Since the only types of alignment objects available are points, lines and circles, all of which require at most two vertices, only two pointers into the vertex list are needed.

All the vertices within the object are stored in the vertex list. Each node stores coordinate information and the rank and id of the vertex. The rank field counts how many times the vertex is referenced by the primitives. This is useful for the cases where a vertex is referenced by more than one primitive. If one of these primitives is deleted, the vertex should not be deleted, whereas if all the primitives are deleted, the vertex should be deleted.

More complex still is the data structure of the image. Simple structures of doubly linked lists are used to maintain the entries in the colour and flag tables. Each node contains the rank of this node, its index into ARIA's colour or flag table, and the value(s) stored in the table.

The image is maintained as a three level data structure. The three levels are the cluster level, the instance level, and the primitive level. The entire structure is very much like that of the alignment object structure except for the instance level. An example of one image cluster is shown in Figure 3.2.

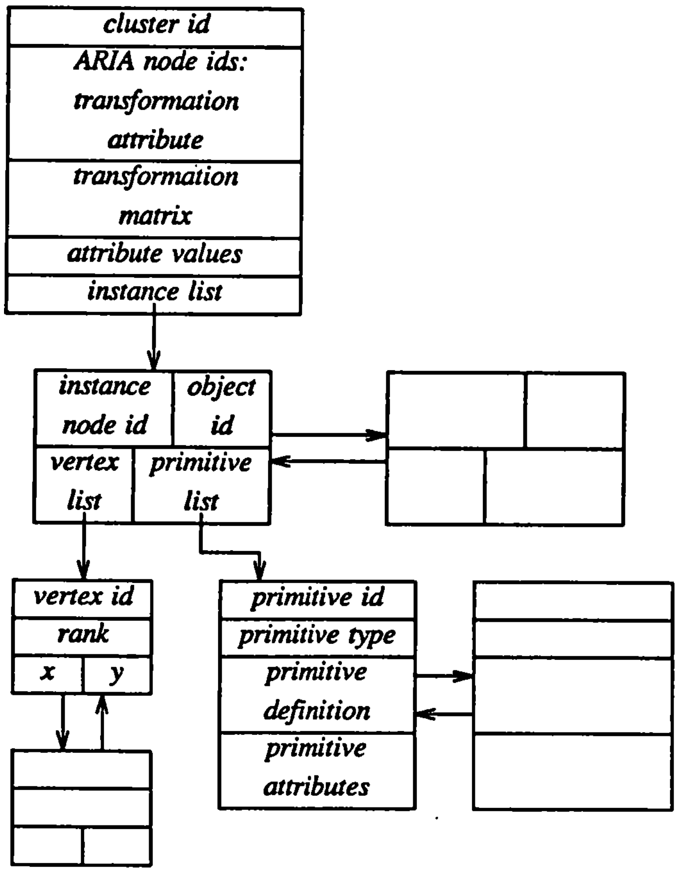

Figure 3.2 — Structure of an image cluster

Each cluster node contains a cluster id, a transformation and attribute ARIA node id, the parameters within the attribute and transformation nodes, and a pointer to a list of instances. Clusters are joined in a doubly linked list.

The instances list corresponds directly with the list of instance nodes in the display list. Each node in the image data structure stores the ARIA instance and object ids as well as pointers to lists of primitives and vertices which comprise the object. The vertex list has the same format as the list used for alignment objects.

Primitive nodes for the image are arranged in a different way than primitive nodes for alignment objects since many more primitive types are used. The primitive type (e.g., line, spline, polygon, etc.), id, definition (i.e., pointers to vertices), and attributes (e.g., line width, pointers into the colour and flag lists, etc.) are stored in a node. The definition and attribute fields are *variant records*. This means that in the space occupied by the definition field, for example, the definition of any primitive may be placed, depending on the primitive type of the node. Thus, the definition field will contain only one vertex pointer for a point primitive, and three vertex pointers for an ellipse primitive. Spline and polygon definitions have to be treated differently since they have a variable number of vertex pointers. In these cases the definition field will contain a pointer to a linked list of nodes, each of which contains a pointer to a vertex and pointer to the next element in the list. The attribute field does not require pointers to lists of entities since attributes for any primitive always use a constant amount of space.

To correspond with the initial configuration of the display list, the initial configuration of the three data structures is now described.

The window list will contain one node with all the information concerning the initial window in the display list. The flags in the node are set so that the window is displayed and alignment objects are not to be included in the picture.

The alignment object's structure will consist of the alignment object's attribute node containing the identical information as the node in the display list, and a cluster node in which every field is filled except for the object type. The primitive and vertex lists will be empty.

The image data structure will consist of one cluster node whose fields are filled, and one instance node comprising the cluster's instance list. The vertex and primitive lists are initially empty.

A point is defined by a single vertex. A line is defined by two vertices, one at each endpoint. A circle is also defined by two vertices, both on the circle and at opposite ends of a diameter. An ellipse is defined by three vertices on its curve. Two of the vertices are at opposite ends of a line drawn through the centre of the ellipse, and the third is one endpoint of the line perpendicular to this first line. Polygons are defined by any number of vertices greater than two, at each intersection of its edges. Splines are defined by any number of vertices greater than three. Each vertex represents a control point for a B-spline curve.

3.5. Task Structure

An application such as the one described in this chapter naturally lends itself to a multitasking approach, due to the many instances of scarce resources [Plebon82b]. The scarce resources that need to be properly managed in this implementation are the frame buffer memory, the display list, the screen space, the geometric data structures, the screen tracker and input devices. The first two resources are the responsibility of the *ARIA Server*, while the rest are the responsibilities of the *Graphics Server, Data Structure Server, Tracker Server*, and a server for each device respectively. This program has an anthropomorphic design, assigning independent processes to roles which might otherwise be considered in a human context.

Adagio was originally a workstation being developed to support research in robotics [MacKay86]. It is a window-based system that provides for rich interaction [Tanner86]. Each window in the system is simultaneously active, in that an input-active task is associated with every window (except image windows which are treated as a single window). The rich interaction is obtained by a kind of parallelism rarely found in other systems.

3.5.1. Devices

Many devices can be simultaneously used in the Adagio workstation. In most configurations a tablet would be used as the primary input device. For this reason the multitasking implementation of the *Tablet Server* is described in this section. Other devices are implemented in a very similar fashion.

The task structure for a tablet is shown in Figure 3.3. In this diagram, and in following diagrams, tasks are represented as ellipses. If task A sends a message to task B, an arrow is drawn in the diagram from task A to task B.

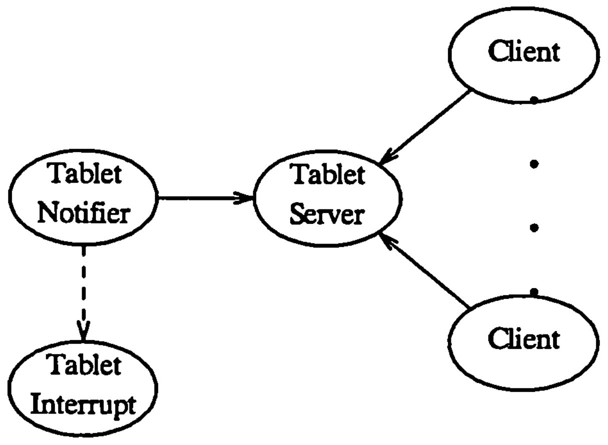

Figure 3.3 – The Tablet Server

The server provides virtual tablets to a number of client tasks where the physical rectangular window associated with this virtual tablet is assigned by the window manager, or Graphics Server. The virtual tablet provides values in the range [0..16383] in *x* and *y* directions. If a virtual tablet corresponds to an image window, this range of values is transformed into image space by the client task for this window. The tablet server also provides tablet statuses (i.e., if the puck is raised or on the tablet surface, etc.). Naturally, the server must also be aware of the positions of the current windows on the screen.

A client task is created for every window, and requests information for that window. It waits for information from the server. Information is sent to a client whenever the puck position on the tablet is within the client's virtual tablet.

The tablet interrupt handler reads hardware registers for the tablet. The tablet notifier constantly reads the information generated by the interrupt handler and accepts a value whenever it is valid and has changed significantly over the previous reading. It thus acts as a filter. Valid values are then sent to the tablet server.

3.5.2. Tracker Server

The tracker server controls the positioning and display of the screen tracker. It provides feedback as to the current *x,y* position of the tracking input device(s). A data structure containing the definitions of all icons and windows is kept by this server. Thus, different tracking icons can be used for each window. Tasks may also change the current icon if, for example, a different mode is entered, such as line drawing mode.

The way this server is implemented is depicted in Figure 3.4. Other devices may also control the tracker with a similar arrangement. The tracker server communicates with the tablet server through a tablet courier. This task simply forwards data to the tracker server. The information passed consists of the screen position, the window in which the tracker currently resides, and the tablet status.

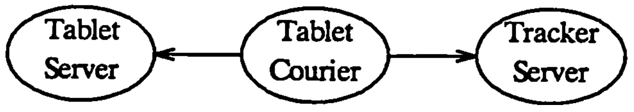

Figure 3.4 — The Tracker Server

3.5.3. Switchboard

The design of Adagio is based on the concept of a ‘switchboard’. On one side of the switchboard are producers of input device data and on the other side are consumers of this data. The consumers request input from the switchboard which in turn routes to these tasks input from devices to which they are connected. The situation is depicted in Figure 3.5. Here couriers are the producers and clients are the consumers.

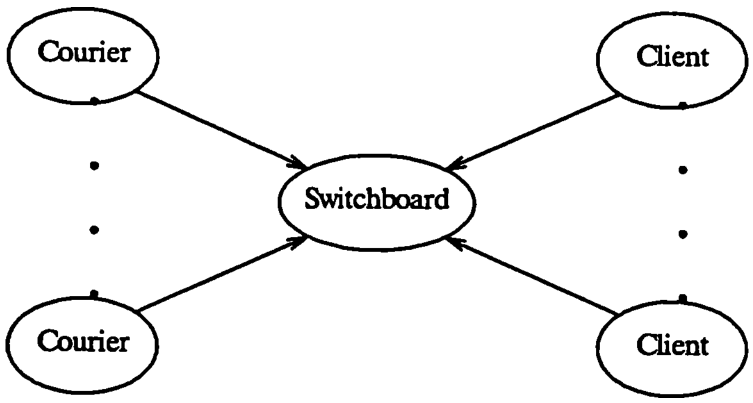

Figure 3.5 — The Switchboard

Couriers provide a link between the switchboard and the device servers. For this application couriers are identical to the client tasks of Figure 3.3. The clients are application tasks in that they use information from the switchboard to carry out some assigned function. In this application the clients will be tasks responsible for windows. The switchboard manages the connections between clients and couriers so that when a client requests a connection to a device, the switchboard creates the courier task and establishes the connection. It also deletes these couriers after the client dies. In addition to these functions, the switchboard also creates tasks for menus when items are selected from them, and initialises the application on system startup (i.e., it creates the menu and image windows).

3.5.4. ARIA Server

The ARIA server is responsible for atomic operations on the display list through ARIA routines and with communicating with the BPS. All requests to write into the displayed portions of the frame buffer memory, besides text operations, are channelled through this server. A task wishing to call an ARIA routine sends a message to the ARIA server with all the necessary values. ARIA is initialised soon after this server is created.

Requests to this server from application tasks are classified within two separate categories. First, a request may or may not cause the image on the screen to change. A request that wouldn't change the image is creating a new node in the display list. Since it could not have been included into the display list without first linking it, there is no need to redraw the image. The second category is whether the ARIA routine returns a meaningful value or not. Information such as primitive or node id's returned from routines should be made known to the requesting task. The value returned can be sent to this task by including it as part of the 'reply' structure. A routine which returns an unmeaningful value does not have to have this value returned to the requesting task. In such cases, a reply can be sent to the task even before the routine is actually called.

Whenever the image should be redrawn, the ARIA server informs the BPS. It also replies to a notifier task that it had created earlier. The notifier delays itself for a short period of time and then checks if the BPS is done. If not, it repeats this process until the BPS is finished. When the image is drawn, the notifier sends a message back to the ARIA server. The BPS can then be started again if necessary. During the time in which the BPS is traversing the display list, the server adds any incoming requests to a queue. Once the notifier message is received, this list is processed and the BPS started again.

Two useful requests that the ARIA server recognises are 'Take Control' and 'Relinquish Control'. If a task sends a 'Take Control' message to the ARIA server, the currently queued requests are first processed and then 'control' over ARIA is granted to the requesting task. During this time this task may call ARIA routines directly since all other requests to the ARIA server are queued and will not be processed until a 'Relinquish Control' message is sent from the controlling task to the ARIA server. When control is relinquished, the queue of requests is processed. A priority scheme among tasks requesting control can be implemented so that higher priority tasks are granted control first even if a lower priority task occurs earlier in the queue. Control may be desired by tasks that wish to provide quick feedback or tasks that do extensive manipulation to the display list and do not want the BPS to be started since it is likely to encounter some inconsistency with the data.

3.5.5. Graphics Server

This server might more appropriately be called the screen or window server since it is responsible for the display of windows on the screen. It receives requests to create, delete or modify windows, and performs the necessary actions. It has layouts for every new type of window that can be created (i.e., property sheets, input windows, etc.). The writing of text into the frame buffer's overlay planes is also done by this server.

The window manipulation requests contain values in the virtual coordinate system of the requestor. The graphics server translates these values into screen coordinates. A data structure containing descriptions of all the windows is maintained. Using this information, the server manages the screen space, and temporarily erases various windows if necessary. Upon receipt of a window request, the appropriate changes in the data structure and display list are effected with messages to the ARIA server. The details of the operation are then passed on to the switchboard in order to alter the couriers or to add or delete tasks representing these windows.

3.5.6. Data Structure Server

This server is the most important task in this application. It is responsible for manipulating and preventing the corruption of application data structures such as the image structure, the alignment object structure, the colour table, and the flags table. The current selection from the image windows is always maintained. Communications with other tasks is shown in Figure 3.6.

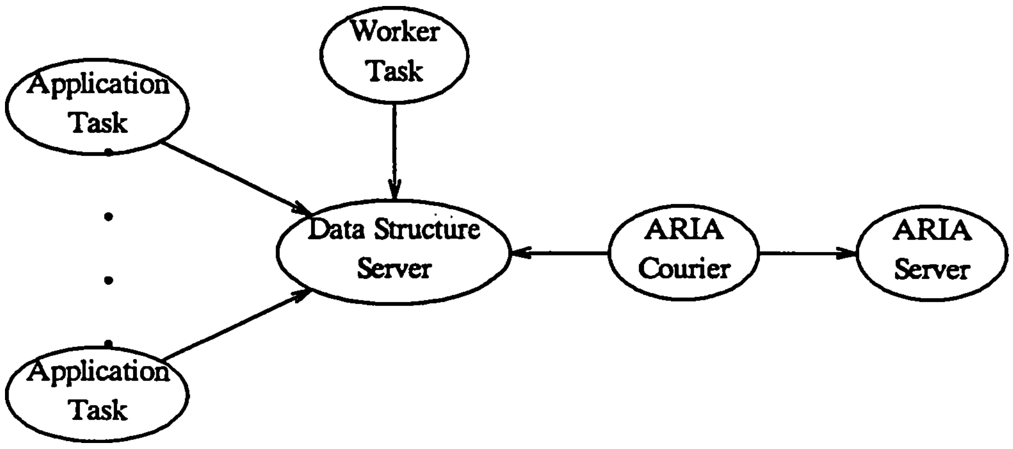

Figure 3.6 — The Data Structure Server

Requests are sent to the data structure server from application tasks to update a data structure or to obtain information from a data structure. If a request requires a change to the display list, a request is forwarded to the ARIA server by an ARIA courier. Requests that require a great deal of processing are satisfied by worker tasks, created by the server.

Each type of request to the data structure server requires a procedure call to be made or a worker task to be created. Short and simple requests are performed by a procedure call within the server itself, and then the requesting task is replied to. Longer requests are handled by creating a worker task which will perform the necessary operations. The requesting task can be replied to while the worker task is active, freeing the requestor for other actions. However, no other operations besides information requests should be performed on the data structure until the worker is complete. Requests that are received in this interval are queued and processed when the worker task finishes.

The operations performed by this server were listed in previous sections. Besides the various operations and queries on the data structures, this server is responsible for the saving and retrieval of image files, and the saving and restoration of session files.

3.5.7. Windows and Menus

As previously mentioned, a single task is responsible for each window on the screen, other than the image windows which are controlled by a single task. Windows other than image windows are system menus or pop-up windows. For simplicity, they will both be referred to as 'menus'. These tasks obtain information from the switchboard in order to carry out some assigned function. In Figure 3.5, they are represented as the 'Client' tasks. Figure 3.7 shows a sample configuration of these tasks for this application. Other tasks such as the Graphics Server and ARIA Server have not been included in this diagram to avoid confusion.

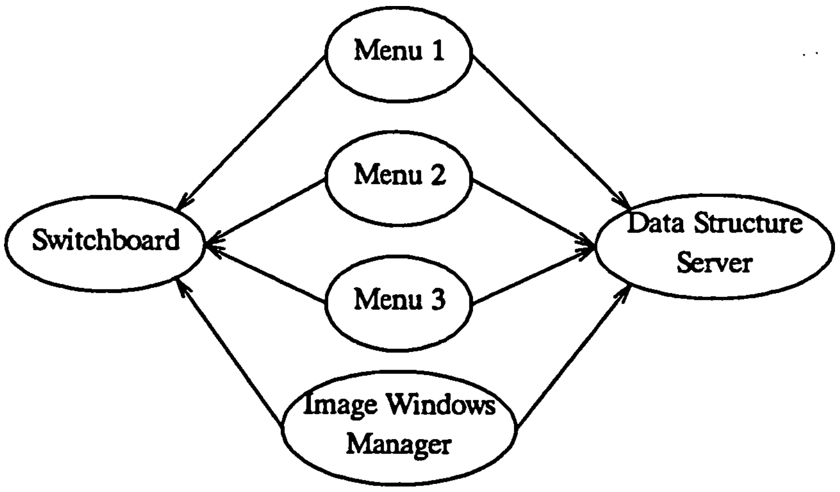

Figure 3.7 – Menus and Image Windows

The menu tasks are created initially by the switchboard. Each task will then send a 'create window' request to the graphics server. The graphics server will draw the menus at the specified location, and will also inform the switchboard of these tasks. The switchboard will then create the necessary virtual tablets for the menus and establish the proper connections. After the graphics server draws the menu, it returns information concerning the locations of its menu items.

The menu tasks are responsible for invoking the correct tasks or sending information to the correct server whenever a selection is made within its window. The image window manager controls 'select' operations within the image windows and with the redefinition of these windows.

3.5.8. VAX Server

The VAX Server is in charge of which tasks can use the VAX serial port and the quadrant of the Ikonas frame buffer that is used for communication with the VAX. This server ensures that only one task has control over these resources at any one time. If a task wishes to use these resources, it sends a request to the VAX server. If no other task currently has these resources, they are granted to the requesting task. The task has to return the resources to the VAX server when it is finished with them by sending another message. If the resources are being used, the requests for them are queued up and are processed in the order they arrive.

3.5.9. EHVT Server

The Echo Handshake Virtual Terminal (EHVT) Server accepts and processes input from the serial port of the Ikonas. Specifically, it acts as a file server on the VAX, or a remote terminal connected to the VAX. This server is responsible for creating new Unix files and reading from existing files, depending on the operation desired.

When the data structure server desires to communicate with the VAX in order to read or write a file, it first requests the VAX resources from the VAX server. When these are obtained, it may communicate with the EHVT server. To save an image or session, the data is first written into the frame buffer memory. Then a write request is sent to the EHVT server, which then reads the data from the frame buffer and writes it into a Unix file. To read the contents of an existing Unix file, a read request is first sent to the EHVT server which then writes the contents of the file into the frame buffer. The data is then read from the frame buffer by the data structure server.

3.6. Improvements and State of Development

The implementation proposed in this chapter can accurately create essentially any two-dimensional drawing. However, the operations the user would use to construct a drawing may be improved. It is also possible that the algorithms used be the application to perform a task may not be optimum.

This section lists some suggested improvements to the implementation described in this chapter. These improvements can be grouped into three areas: ARIA and Ikonas modifications, screen manipulation techniques, and changes in style. Next, the changes necessary to extend the model to 'two-and-a-half' dimensions are listed. Finally, the state of development of this package at the time of writing of this essay is detailed.

Due to restrictions with ARIA and/or the Ikonas, many useful features have either not been implemented or have been implemented in less efficient ways. The following list mentions many changes that would be desirable for this system:

- Currently only 16K of scratchpad memory (SPAD) is available, although additional boards can be added to increase this size to 64K. This is used for communications between processors and for storage of the display list. With a complicated picture and multiple windows, the available memory can easily be used.
- The Ikonas used for this implementation does not contain a character generator which is available on other similar systems. For this reason, ARIA does not support text, and the application has to draw the text in the frame buffer's overlay planes. With a character generator, text could be added to ARIA as a type of display primitive. Text of various fonts, sizes, colours

and directions could be easily implemented with the appropriate routines. Fonts would be stored in SPAD.

- Two modes of display of the image memory are supported by the Ikonas. These are LORES mode (1024x1024x24-bit) and HIRES mode (2048x2048x8-bit). ARIA assumes LORES mode for all its operations. A different implementation of ARIA that uses HIRES mode would be useful, as the resolution is currently only 512x512 in each quadrant. HIRES mode would double the screen resolution. The number of colours available to this application is not nearly as important as in a paint system, for example. Therefore, 8 bits/pixel, or 256 colours is not overly restrictive.
- To be able to handle the increased number of primitives and attributes, much more BPS microcode needs to be written for ARIA. It is extremely unlikely that all of the proposed changes can be implemented in the 64K microcode store. However, this is all the addressable memory available in the current Ikonas configuration.
- Additional ARIA primitives might be useful for some situations. Examples are text, as previously mentioned, bitmaps, and different spline types.
- Each entry of ARIA's flags table currently contains eight bits, each one defining an attribute. Other attributes such as anti-aliasing, stroke width and stroke type are added to the primitive description block of each primitive. It would be useful to include these three attributes in a second flags table. Each entry would comprise up to 16 bits of information. Instead of allowing 350 entries for the colour and flags table, fewer entries could be used and the freed space used for the second flags table. To create two entries for the new table, one entry needs to be deleted from both the colour and flags tables. A suggested arrangement would have 250 entries in the colour and flags tables and 200 entries in the new table.
- The attribute nodes should be allowed to change the stroke width, stroke type, and aliasing in the display list or those nodes beneath it in the DAG. Such an alteration would be made easier if the previous suggestion is implemented.

Various techniques of screen manipulation in this implementation could be added or improved. A list of suggested improvements follows:

- A user should be able to create a window and manipulate either a subset of the clusters of the current image or a new cluster altogether. The user would essentially be operating on two image structures. A cluster, or group of clusters, created in a separate window should then be able to be merged into another window. This abstraction could also be extended to include alignment objects such that different sets of objects are used in different windows. New data structures and manipulation routines for these structures, as well as the display list are needed. More menu options would be required to define and merge these windows.

- As a first step, the alignment objects are implemented as a grid of objects. Subsequent versions of this implementation might allow for objects to be created only at user-specified points, or using heuristics while drawing, or both, as is done in Gargoyle.
- When measuring distances and angles, special icons such as a ruler and protractor respectively could be written into the overlay planes and positioned and manipulated within a window. They would be used in much the same way as rulers and protractors are used in paper and pencil drawings. Whether this is a useful option or not is questionable.

Each user of this system will likely have some screen layout that he would like to use instead of the default system layout. To allow for this, each user may have his own Unix text file describing this layout. A suggested method of allowing individual user layouts is for the command which is invoked to start up the system to first search the home directory of the user for a file called '`.profdraw`'. If this is found, its contents are used to control the setup for this system. Otherwise, the default setup is used.

Instead of having the user enter the full Unix pathnames for every image or session file accessed, a default directory for each should be able to be specified. These can be specified in the command line used to invoke this system. The default location for these directories is the current directory.

If the user would like to access images in other directories while using the system, a possible solution would be to allow the creation of a window which runs a Unix shell or emulator so that he can move around Unix directories and list the contents of these directories.

Another factor which a user may want to control is the definition of various controls on different input devices. He may want to define a button box so that each button represents a function of the fixed or alignment menus. These menus may then be iconified, making more room for the image. A similar approach could be taken with pucks or mice. If, for example, a puck has four buttons, he may define one to perform the 'select' function, one to perform the 'undo' function and one to perform the 'delete' function. The fourth may be defined as an often used operation such as 'show properties', 'add window', or another option not currently available called 'help'. A 'help' selection followed by a selection of some operation would display a window containing a description of that operation.

The changes necessary to extend this implementation to 'two-and-a-half' dimensions are not extensive. Since ARIA is designed as a three dimensional graphics package, no changes to it are necessary since perspective projections are available and z-buffering is done. A *polygonal object* primitive type is already implemented in ARIA. However, if spline surfaces are needed, they have to be added.

The most difficult part of this extension is the use of input devices to control a third degree of freedom (i.e., the *z* values). A few exotic devices are available which are capable of doing this, but the same input devices as were used in the two dimensional case are assumed to be the only ones available. A common method of inputting three dimensional data with a device that has two degrees of freedom is to use three windows, one of each which shows the view of the image looking from a point on each of the positive coordinate axes towards the origin, assuming the image is located at the origin. The user then manipulates a position marker which is displayed in all three windows. Each window can control the location of this marker in two dimensions. Combining the three windows gives the user the ability to move around in three dimensions. The same manipulations can be performed by using only two of the three windows. The third window is included as an extra aid. Frequently, a perspective view of the image is simultaneously shown in a fourth window.

As far as the application is concerned, every instance of a coordinate pair must now be changed to a triplet. Additional functionality would be needed in order for a user to control added attributes such as *z*-buffering, and to be able to add the new primitive types.

The implementation that has been described in this chapter is an example of a multitasked design of a two dimensional draw package. This design has been partially implemented in the Computer Graphics Lab at the University of Waterloo, although much of the application code still needs to be written.

Harmony is supported by the National Research Council (NRC) of Canada, and contains a number of device drivers and servers. These servers perform as specified and have been fully implemented.

Adagio was also conceived and developed at NRC but its performance has not yet been fully tested with an application. However, no problems have yet to be found with it. This includes the switchboard, the tablet couriers, simple menu tasks and image windows.

ARIA is currently capable of its original drawing primitives and attributes plus an anti-aliasing option. Lines of variable width are being tested as are spline curves.

The ARIA server has been largely implemented. However, the interface needed to pass a structure of information generated by three ARIA routines has not been developed. The graphics server is able to create and destroy windows and inform the switchboard of these operations. It maintains a window data structure but is not able to distinguish overlapping windows. It is not able to change the definitions of the current windows and has no knowledge of special windows such as property sheets. The data structure server is in a low state of development. A user is currently able to add only points, lines and polygons but is not able to manipulate them in any way. The software described in this paragraph has been written largely by the author. Initially, the data structure server and the graphics server were written at the National Research Council, but have undergone extensive modifications.

References

[Baudelaire75] Patrick C. Baudelaire, "Draw" video tape, Xerox PARC, September 1975.

[Baudelaire79] Patrick C. Baudelaire,"Draw Manual", *Alto User's Handbook*, Xerox Corp., 1979.

[Bier86] Eric A. Bier and Maureen C. Stone, "Snap-dragging", *Computer Graphics*, 1986, pp. 233-240.

[Booth82] K.S. Booth, W.M. Gentleman and J. Schaeffer, "Anthropomorphic Programming", Technical Report CS-82-47, Dept. of Computer Science, University of Waterloo, May 1982.

[Booth85] K.S. Booth, W.B. Cowan and D.R. Forsey, "Multitasking Support in a Graphics Workstation", *Proc. 1st International Conference and Exhibition on Computer Workstations*, San Jose, Calif., Nov. 1985, pp. 82-90.

[Borning86] Alan Borning, "Defining Constraints Graphically", *CHI '86 Conference Proceedings*, April 1986, pp. 137-143.

[Burden81] Richard L. Burden, J. Douglas Faires and Albert C. Reynolds, "Numerical Analysis, Second Edition", pp. 107-119, July 1981.

[Buxton82] W. Buxton, S. Patel, W. Reeves and R. Baecker, "OBJED and the Design of Timbral Resources", *Computer Music Journal*, 6(2), 1982, pp. 32-44.

[Englebart68] D.C. Englebart and W.K. English, "A research center for augmenting human intellect", *AFIPS Conference Proceedings*, Vol. 33, 1968, pp. 395-410.

[Evans81] K. Evans, P. Tanner and M. Wein, "Tablet-based valuators that provide one, two, or three degrees of freedom", *Computer Graphics*, 15(3), 1981, pp. 91-97.

[Forsey85] D.R. Forsey, "Harmony in transposition: A toccata for VAX and Motorolla 68000", M.Math thesis, University of Waterloo, 1985.

[Gardan84] Yvon Gardan and Michel Lucas, "Interactive Graphics in CAD", 1984.

[Gentleman81] W.M. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept", *Software Practice and Experience*, Vol. 11, pp. 436-466, 1981.

[Gentleman85] W.M. Gentleman, "Using the Harmony Operating System", Report of DEE, NRC of Canada NRCC-ERB-966, Ottawa, Ontario, Dec. 1983, revised May 1985.

[Gurd83] R.P. Gurd, "A Microcode C Compiler for the Bit-Sliced Microprocessor", M.Math thesis, Department of Computer Science, University of Waterloo, 1983.

[Gurd85] R.P. Gurd, *Microcode C Language Summary Ikonas Version 4.8*, 1985.

[Ikonas82] Ikonas Graphics Systems, Inc. (a subsidiary of Adage, Inc.), *RDS-3000 User's Guide*, Part No. 10-301-095-10A, Billerica, Massachusetts, April 1982.

[Kernighan81] B.W. Kernighan, "PIC – A Language for Typesetting Graphics", *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, Vol. 16, No. 6, June 1981, pp. 92-98.

[Knuth79] D.E. Knuth, "METAFONT, a system for alphabet design", *TEX and METAFONT: New Directions in Typesetting,* American Mathematical Society and Digital Press, 1979, Chapter 3.

[Laidlaw86] D.H. Laidlaw, W.B. Trumbore and J.F. Hughes, "Constructive Solid Geometry for Polyhedral Objects", *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 161-170.

[Lampson79] B.W. Lampson, "Bravo Manual", *Alto User's Handbook,* Xerox PARC, 1979, pp. 31-62.

[Lipkie82] Daniel E. Lipkie, Steven R. Evans, John K. Newlin and Robert L. Weissman, "Star Graphics: An Object Oriented Implementation", Xerox Corp., *Computer Graphics*, July '82, pp. 115-124.

[Loo86] Ruby Loo, "ARIA – A Near-Real-Time Graphics Package", M.Math thesis, University of Waterloo, 1986.

[MacDraw84] MacDraw Manual, Apple Computer, Inc., 1984.

[MacKay86] S.A. MacKay and P.P. Tanner, "Graphics Tools in Adagio, a Robotics Multitasking Multiprocessor Workstation", *Graphics Interface '86*, Vancouver, May 1986, pp.98-103.

[Nelson85a] Greg Nelson, "Juno, a constraint-based graphics system", *Computer Graphics*, 19(3):235-243, July 1985.

[Nelson85b] Greg Nelson, "Juno", video tape, CHI '85 Highlights, included in *Siggraph Video Review #19*, 1985.

[Newman75] William Newman, "Markup" video tape, Xerox PARC, June 1975.

[Olsen86] Dan Olsen, Jr., "Editing Templates: A User Interface Generation Tool", *Computer Graphics and Applications*, November 1986, pp. 40-45.

[Opperman84] Mark Opperman, "A Gremlin Tutorial for the SUN Workstation", Internal document, EECS Department, UC Berkeley.

[Ossanna79] J.F. Ossanna, "NROFF/TROFF User's Manual", *UNIX Programmer's Manual*, 2 (January 1979), Section 22.

[Pavlidis85] Theo Pavlidis, Christopher J. Van Wyk, "An Automatic Beautifier for Drawings and Illustrations", *Computer Graphics*, July '85, pp. 225-235.

[Plebon82a] Darlene A. Plebon and Kellogg S. Booth, "Interactive Picture Creation Systems", University of Waterloo Technical Report CS-82-46, December 1982.

[Plebon82b] R.J. Beach, J.C. Beatty, K.S. Booth, E.L. Fiume and D.A. Plebon, "The message is the medium: Multiprocess structuring of an interactive paint program", *Computer Graphics*, Vol. 16, No. 3, pp. 277-287, July 1982.

[Requicha80] A.A.G. Requicha, "Representations of Rigid Solid Objects", CREST Advanced Course, Darmstadt, September 1980, pp. 2-78.

[Smith78] A.R. Smith, "Paint", Technical Memo. No. 7, Computer Graphics Lab, NYIT, Old Westbury, NY, July 1978.

[Stone80a] Maureen Stone, "How to use Griffin", Internal memo, Xerox PARC, 1980.

[Stone80b] Maureen Stone, "Griffin", video tape, Xerox PARC, 1980.

[Sutherland63] I.E. Sutherland, "Sketchpad: A man-machine Graphical Communication System", Technical Report No. 296, Lincoln Laboratory, M.I.T., January 1963.

[Sutherland84] I.E. Sutherland, "Sketchpad", video tape, included in *Siggraph Video Review #13*, October 1984.

[Tanner85] P.P. Tanner, M. Wein, W.M. Gentleman, S.A. MacKay and D.A. Stewart, "The User Interface of Adagio, A Robotics Multitasking Multiprocessor Workstation", *Proc. 1st International Conference on Computer Workstations*, pp. 90-98, San Jose, Nov. 1985.

[Tanner86] P.P. Tanner, S.A. MacKay, D.A. Stewart and M. Wein, "A Multitasking Switchboard Approach to User Interface Management", *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 241-248.

[Teitelman84] Warren Teitelman, *The Cedar Programming Environment: A Midterm Report and Examination,* Xerox PARC Technical Report, CSL-83-11, June 1984.

[Thomas83] James Thomas and Griffith Hamlin, "Graphical Input Interaction Technique (GITT) Workshop Summary", *Computer Graphics*, 17(1), January 1983.

[VanWyk82] C.J. Van Wyk, "A High-Level Language for Specifying Pictures", *Transactions on Graphics*, ACM, Vol. 1, No. 2, April 1982, pp. 163-182.

[Vernadat84] F.B. Vernadat, "A Commented and Indexed Bibliography on Data Structuring and Data Management in CAD/CAM: 1970 to MID-1983", National Research Council of Canada report ERB-956, March 1984.

[Wein81] M. Wein, P.P. Tanner and K.B. Evans, "Graphic Interaction at NRC", video tape, included in *Siggraph Video Review #4*, 1981.