

A Conceptual Discussion for an
AIRBORNE GRAPHICS SOFTWARE SUPPORT SYSTEM

Derryl A. Williams

Air Force Wright Aeronautical Laboratories
Wright-Patterson Air Force Base, Ohio

Abstract:

This paper describes a concept for a software support system for the development of airborne graphics software. The advent of the CRT in cockpits and the current trend toward complex pictorial formats has caused a significant increase in the amount of graphics software on board the aircraft. However, current software development tools rely on hand coding and line-by-line generation. These techniques do not lend themselves to the pictorial or artistic nature of digital graphics. The software support system discussed in this paper provides several key items. First, an Interactive Workstation allows the designer to essentially "draw" the desired image and specify the dynamic attributes. A Code Generation System then creates hardware independent source code. Finally, a Targetting Compiler section creates hardware specific code and also modifies existing system code to accept the new graphics module. This automated system concept not only supports the initial graphics software creation but also supports the software maintenance of fielded systems. The system allows rapid production of software and significantly reduces the software skills required of the user.

Background:

Airborne display generation system requirements are becoming more and more complex and the need for support systems has become evident. In order to design such a support system, some assumptions and observations must be made of airborne systems and their operation.

In the simplest terms, the function of a display generation system is to accept data from the aircraft system and display it in a graphical representation. This graphical representation may be as simple as an alpha-numeric display of the data or as complex as a pictorial image distilled from several parameters. One aspect, however, is that the image (simple or complex) represents a snap-shot of the input data such that any dynamics of the display are explicitly controlled by the input data and are not implied, predicted or otherwise imparted by the display generation system autonomously. The significance of this fact is that, although display generation architectures (current and future) may vary, the interaction of

the display generation system with the rest of the avionics system will follow this same basic design philosophy.

A trend is emerging in hardware architectures as well. The display generation system is often composed of two functional elements (see Figure 1). The first element is a general purpose processor which acts as the interface to the rest of the avionics system often via a multiplex avionics bus as shown in Figure 1. The general purpose processor also functions as a host and controller for the second element -- the graphics engine. The partitioning of tasks between these two elements may vary from system to system depending on hardware capabilities, the basic architecture is generally evident. The design of the software support system in this paper assumes these operational and hardware architecture philosophies.

Introduction:

The ultimate purpose of a support system is to produce software code that, when executed on a display generation system, produces the desired image. Stated even more simply, the task is to encode an image in software. As such, the creative aspects of the process are artistic or pictorial in nature. Whether the picture is a two dimensional image or a two dimensional representation of a three dimensional image, the design process deals with the image and the spatial relationships of the elements composing the image. Clearly, this is much removed from the sequential lines of code of conventional programming methods. As a minimum, any image is at least two-dimensional whereas software coding is one-dimensional. Furthermore, sequential coding's single dimension can be mapped easily to time but not space. Clearly, conventional coding techniques are highly inefficient for graphics work yet current systems require the graphics programmer to code in this serial manner. The spatial relations of the objects must be translated by the programmer into numbers in a display coordinate system. Once coded into digits, the intuitive spatial relationships are lost and the programmer must devote considerable energy to insuring that the data is correct. Compounding this problem is the fact that coordinate systems can be changed drastically effecting the resulting image. The biggest problem of this process is that the final image must be visualized in the mind through what amounts to mentally compiling the

Avionics System Architecture

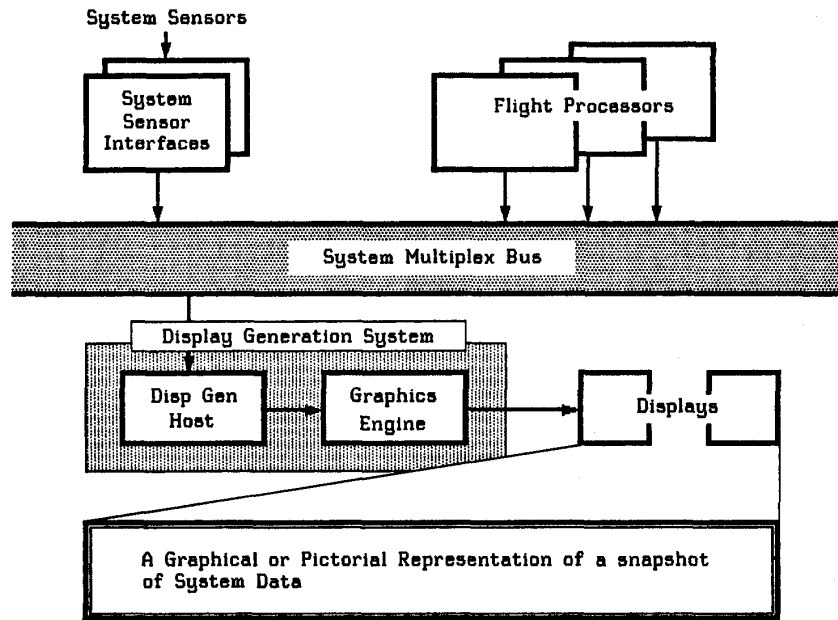


Figure 1

code. The programmer can not actually see the image as it evolves. A better method must be provided to not only allow the programmer to see the evolving image but also to automatically generate code and verify data thus leaving the user free to devote all energy to the creative aspects of the task. It is for this reason that a critical aspect of this proposed system is to provide a user interface tailored to pictorial image design. The designer must see the results of his inputs pictorially and immediately. Without such an interactive interface display software coding could be likened to painting a picture in the dark.

The proposed system has been divided into several functional modules. Figure 2 is a block diagram of these modules and their relationships. Also indicated are the higher level functional area groupings of the system. These areas are: the Interactive Workstation where all user interaction takes place, the Code Generation System producing source code for the graphics engine and the host processor, and the Target Compilers converting the source code to machine code for the specific target system. The final products of the system are the three software load modules ready to be loaded into the aircraft systems. Following is a description of each area and its functional elements. It should be noted that the block diagram and the discussion are based on a conceptual functional

partitioning. The diagram is intended to imply a desired level of autonomy among system elements. In reality, the actual implementation of such a system would require much more interaction among processes as a practical matter. Particularly if real-time operation is to be realized. In this sense the discussion represents the idealized goal for which to strive.

Interactive Workstation:

The Interactive Workstation is where the designer interacts with the system. The goal of this workstation is to support the image design process. As stated earlier, the design process is pictorial or spatial in nature. Accordingly, the method of interaction at this workstation is spatial. The designer is provided with input devices such as joysticks, digitizing tablets, and mice for spatial inputs as well as a keyboard for specific data or other alpha-numeric inputs. For feedback a CRT screen is provided to display the evolving image. It is critical that this display be updated in real time as the design process progresses. The designer is operating with an image on the screen and need not get involved with the "lines of code" of more conventional software development methods.

Airborne Graphics Software Support System

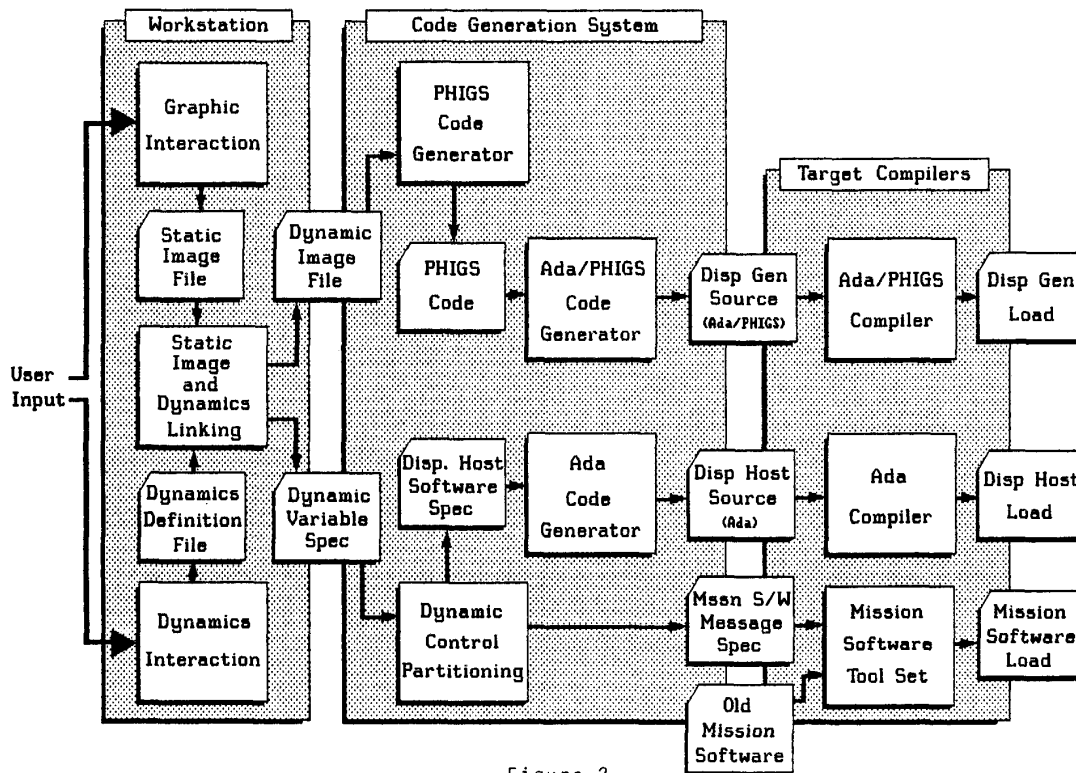


Figure 2

Since no code is presented directly to the user at this point it is not necessary for the final source code to be generated during this process only that the evolving image be coded and stored internally in a form easily edited as the design progresses. This approach allows the user to describe the entire image to the support system before final code is generated. As such, the code generation can then be optimized accordingly.

The interactive support capabilities of the workstation must be robust. The system is designed as an object oriented editor. The user will be able to create objects on the screen and to assemble those objects into the final image. The user will be able to not only initially create the objects on the screen but will also be able to easily edit or even delete objects as desired. The system must accommodate this editing in a natural manner and must not require exact backtracking or require the following of very complex procedures. In short, any object must be able to be edited at any time.

The object oriented approach is vital to establishing the image dynamics. In this system the user need only specify the top level dynamic action (rotate, translate, etc.) as applicable to

the object as a whole and is not required to dissect and supply the specifications to every point or vector primitive composing the object. The ability to edit in this manner will have a strong influence on how the workstation software stores and manipulates the image description.

Since the dynamics of the display are very important, a separate functional block is shown in figure 2 for this process. As discussed earlier the image dynamics are defined at the object level where specific objects are assigned dynamic actions relative to the rest of the image. For example, an object may be designated to move right or left on the screen. For the dynamic action to have meaning, however, it must be commanded by an external input. Therefore, to complete the specification, the dynamic action must be linked or "hooked" to parameters to be input from the aircraft system via the mission software. This linking is critical because, as stated earlier, all dynamics are explicitly controlled by system parameters and none are imparted by the display system autonomously. As such, the dynamic interaction function of the workstation consists of two operations. In the first step, the user specifies the desired action of the object including limits of travel, origins, calibration

points and any other constraints necessary. In the second step, the user specifies the system data to be used including information such as range limits of data and any functional operations required such as scaling or normalization. This second step actually departs from the spatial or artistic operations of the display design process but should be supported by the same workstation hardware and is considered an integral part of the display design process. The result of these operations is a dynamic variable specification block or file. This is separate from the image file since the display generation host processor must be involved with the dynamics.

The data received from mission software is not limited to numerical values. Quite often an object in a display is either activated (visible) or deactivated (invisible) according to specific mission software modes or conditions. The visibility of an object can be considered a dynamic attribute and is thus specified from the workstation and mapped to logical variables or flags. The specifications for such logical variables are included in the dynamic variable specification block.

As discussed earlier, the problem with developing graphics code in the conventional line-by-line method is that the resulting image is not immediately evident. The same sort of problem exists when image dynamics are specified. The designer must mentally reconcile the desired image movement with the specifications for the inputs from mission software. To overcome this problem the support system should include the ability to exercise the dynamics from the workstation. For example, the peripherals such as the joystick normally used for spatial inputs can be reprogrammed to emulate dynamic parameters. Extremely complex image movements involving several variables at once may not be practical but selectively exercising inputs will prove valuable to the designer in verifying the specifications.

The results of the user interaction consists of two files which are then melded into one file for final output. The graphical interaction function produces the Static Image Description file and the dynamics interaction function produces the Image Dynamics Definition file. When combined the two files form the Dynamic Image Description file which includes the mission software data requirements specification. The workstation software may create all these files as separate and autonomous files or a tight cross-linking (thus a blurring of the partitioning) may be required for efficiency. However, since the files would not be accessed by the user directly this would not cause a problem.

Although the output of the graphical interaction function is referred to as a "static" image description, some special structuring is required to eventually accommodate the dynamic hooks. In the normal sequence of events the user first specifies the object then defines the dynamics. In fact the entire display may be designed before any dynamics are specified.

Furthermore, the workstation will also support editing an existing image either graphically or dynamically. Therefore, the static image description must have the provisions in place for dynamic hooks on all objects. If an object is not eventually assigned dynamics the system would merely load constant values in place of the missing variables. Of course, it would not be efficient to have all of these unused hooks in the final airborne software but at this stage it would pose no problems. To accommodate all of the potential dynamics implies creating, as part of the object definition, a set of variables controlling all possible dynamic actions. At first this may seem impractical but, in fact, even complex dynamic actions can be described with a combination of only a few basic actions. Initially, defining objects with all of these basic hooks would not be unreasonable. If the Static Image File is created with this structure then linking the Dynamic Definition File is a straight forward task. The unused hooks can be flagged for later removal by the Code Generation System.

The final outputs of the workstation would be two files: the Dynamic Image Description file and the Dynamic Variable Specification file. The dynamic image file is a file of graphics code for the newly created image. The format need not be human readable at this point although that would be preferred. The code will have all the dynamic variable links in place along with flags on the unused variables. The file would also contain a common block or "compool" definition for the dynamic variables.

The Dynamic Variable Specification file would contain a description or definition of all variables required by the image file. The file would contain a duplicate of the variable common block contained in the image file. The file would also specify the source of all parameters and any required functional or logical derivations of the parameters. These functions may be as simple as scaling or normalization or may be complex functions containing several input parameters. Logical functions often used for such things as display mode changes would be defined here as well. The Dynamic Variable Specification would contain all information required to manipulate and control the image. When paired with the Dynamic Image Description file the two files would contain all the information necessary to specify the image and its operation.

Since the Dynamic Image Description file and the Dynamic Variable Specification file form the inputs to the next section of the system (the Code Generator) this is designated as an interface control point. A strict interface specification must be established and rigidly followed. If interface control is not established, future enhancements such as new hardware systems would require extensive software modification and may be incompatible with software previously generated by the support system. Interface control cannot be over emphasized.

Code Generation System:

The function of the code generation system is to convert the output files from the workstation into machine independent source code. This source code would also be human readable. It is at this point that language standards would be applied. For the graphics code the Programmers Hierarchical Interactive Graphics System (PHIGS) standard is selected for this discussion. PHIGS is not a stand-alone language but is, instead, a graphical language structure specifying types of commands and their interrelationships. To complete the specification the PHIGS definition must be implemented or "bound" with a higher order language (HOL). In this system the HOL chosen is Ada. Both standards must be implemented completely and neither will be violated in any way.

Such a bold statement probably requires some explanation. On the one side, PHIGS and Ada are standards which should not be compromised. On the other side, the purpose of the support system is the generation of real-time graphics. As such, the standards should be implemented completely but, in most cases, only a subset will actually end up being used in the final code. For example, PHIGS has functions for the support CAD/CAM systems and multiple workstation environments which do not exist on aircraft. Furthermore, as shown in Figure 1, the display system is an output only channel with no inputs back to the aircraft system. These factors greatly simplify the requirements and can aid in the support real-time operation. Similarly, some functions of Ada may not be heavily used for graphics code and the graphics engine, although capable of executing these functions, need not be efficient at doing so. The overriding consideration is real-time operation.

The architecture of the code generation system is driven by the aircraft system architecture as discussed earlier. As mentioned, the assumption is made that the aircraft's display generation system consists of a display generator (the graphics engine) and a display generator host (a general purpose processor such as a MIL-STD-1750A). Accordingly, the operation of the code generator accommodates the partitioning of these functions by dividing the code generation task into two parallel operations. The first path implements the image description for the graphics engine and the second path codes the dynamics and control functions for the display generation host.

The first task of the code generator would be to convert the Dynamic Image File into a PHIGS description. This would actually be a pseudo-code and, as such, could not be compiled directly. The PHIGS file is, however, a complete description of the image in an object oriented structure and includes identification of all the dynamic variable inputs. This would be the first generated file that is human readable. The PHIGS file is the input to the Ada/PHIGS code generator.

The Ada/PHIGS code generator would create graphics source code according to the Ada/PHIGS combined standard. The code would be complete and

syntactically correct. Parameter common blocks are included commensurate with the dynamic variable specification. The code would also include user comments if desired.

Along the parallel path, the dynamic variable specification and common block structure are input to the Ada code generator for targetting to the display generator host. The dynamic variable specification may take the form of algebraic equations, state diagram descriptions, or rules of operation (e.g. mode selections). Because of this complexity, the Ada code generation may require user intervention and decisions. The system must, however, automatically take care of details such as code syntax etc. thus requiring only high level inputs from the user.

If the Ada/PHIGS and Ada code were generated by hand, the programmer would follow many rules in the process. Besides the rules of good software coding, there would likely be rules or guidelines governing the partitioning of tasks between the graphics engine and the host processor. These rules would be different for different target systems depending on specific architectures and capabilities. Similarly, the automatic code generators must follow the same rules. This does not necessarily imply an expert system per se, only that a rule base is necessary. In keeping with the modular concept of the system the rule bases will be a separate module such that a different target display system can be accommodated with only a data base change. Such a database is often referred to as "personality module". In the ideal case, all target system personality would be accommodated in the Target Compiler section leaving the Code Generator to be completely hardware independent. However, as a practical matter, the target system's capabilities must be known to efficiently partition the tasks between the host and the graphics engine.

The primary outputs of the code generation system are two source code files. Although the structure and relative partitioning of these files has been influenced by the target system, the files are still, nonetheless, machine independent at this point. The files conform to language standards serving as another interface control specifications. This feature would provide for software transportability.

Since the display generation system must operate in response to inputs from the other aircraft systems, an interface specification is required to define this communication. A third file must be created by the code generation system defining a message to be implemented in the mission software. If it is assumed, for the sake of discussion, that the display generation system is interfaced via a MIL-STD-1553B avionics bus, then the message specification would define a 1553 message format including items such as the required parameters, repetition rate, remote terminal address and subaddresses etc. This message specification file would be formatted so as to be input to a mission software tool set. The tool set would combine the new message requirement with the existing mission software and would create a new software load. Of course, this tool set is far

from a trivial capability and, as such, is not within the scope of this discussion. Suffice to say, if the support tools are not available, the changes to mission software would be made by hand supported by a specification file produced by the Code Generation System. In either case the interface to the mission software is another point for an interface control specification. Such controls would be accommodated via rule bases governing the dynamic variable coding even as far back as the workstation.

Targetting Compilers:

The final section of this system is the Targetting Compilers. The first of these would be a compiler to target Ada/PHIGS to the graphics engine. The critical factor is efficiency. Both Ada and PHIGS are powerful and complex languages, hence, compilers can easily become inefficient and produce code that will not execute in real time. The design of the compiler must consider the execution speed of the generated code as the highest priority. It is required that the compiler support full Ada and PHIGS code but, as discussed earlier, the identification of a "real time subset" may be in order. As hardware capabilities improve, more of Ada and PHIGS can be included in the real-time subset.

In a like manner, the display generator host processor must be supported by an Ada compiler. All the concerns for speed and efficiency apply here as well. The host must not only process the data in real time but must be able to communicate with the system bus and the graphics engine efficiently as well. In this case the speed is not so much limited by the use of Ada as by the specific hardware interface design.

In this discussion the term "compiler" refers to more than a strict definition might imply. Graphics system architectures vary greatly. As such, the most efficient programming commands vary as well. The Target Compiler section is envisioned to contain optimization routines to accommodate the hardware variations. These variations may require several passes to achieve optimization but since this is an off-line task it does not cause a problem. Unfortunately, optimization is more easily described than implemented and will require extensive development efforts.

The final outputs of the compilers and the overall system are three software load modules. The first module is loaded into the graphics engine, the second is loaded into the graphics host. The third load module is the updated mission software for the avionics flight processors.

Summary:

In recent years the capabilities of digital graphics systems have increased dramatically. Future aircraft systems will take advantage of these technology advances by employing more complex displays to improve the pilots situation awareness and hence performance. These improved systems will, however, place an ever increasing burden on development and support systems. The objective of this proposed system is to provide the needed support in an efficient manner. In the past graphics software was generated by hand and required very specialized skills. Furthermore, the image could only be intuitively visualized during the development. This system provides an interactive interface where a designer can artistically or pictorially design the displays. The system also provides a similar interface for the designer to specify the operation of the display. The remainder of the development process is automated thus eliminating the requirement for specialized programming skills. The system not only provides an environment tailored to the design process, but also produces results orders of magnitude faster than before. This capability will be valuable as a system support tool for operational systems as well. The design of the support system is modular and extendable to avoid obsolescence as new hardware system evolve.