A System for Conducting Experiments Concerning Human Factors in Interactive Graphics

L. R. Bartram, K. S. Booth, W. B. Cowan, J. D. Morrison, and P. P. Tanner

Computer Graphics Laboratory
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
519/888-4534

Abstract

This paper describes a system for designing, running, and analyzing experiments in cognitive psychology. The software tools for this are divided into two sets, those that run on a mainframe to support the design and analysis functions and those that run on a dedicated multiprocessor workstation to run the experiments. This division is dictated by a fundamental dichotomy between the need for a rich environment for the design phase and the need for responsiveness in the running phase of the experiments. The motivation for this work is a desire to understand the parameters of human real-time performance in the context of personal computing. The experiments examine temporal, spatial, and chromatic aspects of interactive graphics in workstation environments. The paper provides an overall framework within which these issues are being studied and suggests a methodology for conducting the studies based on the richness vs. responsiveness dichotomy.

1. Introduction

The world is not taking full advantage of the computing power it has, and more is coming. In considering the utility of an interactive graphics system two general features stand out: the desirability of a rich set of powerful tools to handle as wide as possible a range of computational tasks, and the desirability for an environment that can be molded to the perceptual and motor characteristics of the individual user. These issues are particularly interesting in environments in which a single user is closely coupled to a single machine (a personal workstation) since this is the computing environment in which adapting the computer to the human is most likely, in which the greatest increase in effective computer power is possible, and in which the greatest increase in computer usage is occurring. Additional computational power can be used to make an environment more rich and powerful, as has generally been the case with multi-user systems, or to make it more responsive and adaptable, as has generally been the case with real-time systems. But a fundamental opposition seems to make the two qualities difficult to improve simultaneously. This duality of interactive computing leads to the issues dealt with in this paper.

To answer the many questions that present themselves, it is necessary to perform experiments that determine how people use computers as tools. Our own experience doing experiments shows that the very problems we are trying to investigate plague the experimental process – we want to use computers to assist in the design, running, and analysis of experiments, but we don't have a set of tools that are powerful enough and responsive enough to do the job. We have chosen to divide the problem into two parts, and to attack each part separately. Our experiments run on stand-alone workstations based on a multiprocessor operating system kernel for real-time control where we can provide the level of responsiveness we need. The design and analysis of the experiments is accomplished using a rich set of tools running in a mainframe Unix environment where we have a solid base on which to build.

Software for both parts of the experimentation process has been designed and implemented. It is being used to conduct cognition and perception experiments related to our studies of human factors in interactive graphics. Each set of tools has its own requirements and its own approach. The two are related by a common goal, but the philosophy (perhaps "religion" is more accurate) of each is unique. The mainframe-based tools support the design, testing, and analysis of experiments; their emphasis is on richness and robustness of function. The workstation-based tools support the actual running of experiments; their emphasis is on responsiveness and, equally important to our needs, reliability of responsiveness.

We are still refining the two sets of tools. Our experience so far convinces us that this approach to the problem extends to many other problems for which computers are, or could be, used. In particular, the specific lessons that we are learning from building our experiments are easily generalized to a wide range of

---

This work is supported by the Natural Sciences and Engineering Research Council of Canada under Grant #G1579.

personal workstation applications where the need for a rich and powerful set of tools must be accompanied by a close coupling between the user and the computer to achieve the symbiosis of which everyone dreams when they acquire a personal computer.

The next section sets the stage for the rest of the paper by laying out the dichotomy that we see between richness of function and reliability of responsiveness in computer systems. Section 3 discusses the problems that are encountered in designing, running, and analyzing human factors experiments and the very different requirements of the design phase and the running phase of an experiment. Section 4 describes a collection of tools that has been developed to solve these problems and the different techniques employed in the two phases. Section 5 summarizes the main contributions of the research and poses questions for future study. An example of the use of the tools to design and run an experiment concerned with the use of color in interactive graphics applications is presented as an appendix.

2. The Richness vs. Responsiveness Dichotomy

In terms of price and physical size, personal computers have remained constant (to an order of magnitude) throughout their short history. But in terms of any computational measure (MIPS, RAM capacity, disk capacity, etc.) they continue to expand at an amazing rate. What is the purpose of all of this computational power? Surely the answer must be: "to make the personal computer more personal". More precisely, to make it a better complement to its human user.

We consider a computing environment to be *rich* when there are many actions available to a user at any one moment. Unix, for example, is richer than MS-DOS. A full window environment, such as provided by the X Window System or NeWS, is richer than either. An environment lacking in richness will be called *poor*. Rich environments tend to be very powerful, but also very demanding of the user. Thus, the more experience a user gains, the more richness he desires in his environment. Put another way, the more personal a computer becomes, the richer its user will want it to be.

Rich systems tend to have several characteristics in common.

- *Multi-layered:* each new feature extends existing features and uses the earlier features as a building block.
- *Large:* the simultaneous presence of many features and the building block approach imply a lack of economy in resource utilization.
- *Cluttered:* a collection of background tasks provides system support programs that frequently run in parallel with user-invoked programs.

All of these characteristics tend to make rich environments slower and less predictable than poor ones. By contrast, *responsive* environments are those in which the interface between the human and the computer is tailored to human perception and cognition. Three characteristics are typically mentioned as being important.

- *Time:* the environment must offer interaction at a rate suitable to human ability to manage the information density provided by the system.
- *Space:* the environment must provide information over a large enough region of space and with a high enough density to be optimally manageable by the human visual system.
- *Color:* the environment must provide the right color at the right place and the right time.

Non-visual characteristics, auditory and tactile, are also important, as is the coordination of the various modalities. Appropriate temporal, spatial, and chromatic relationships between incoming and outgoing information are essential features of a responsive system. Because predictability and precision are at the heart of responsive systems, such systems tend to be small, with few features and few things happening at one time.

Our definition of responsiveness differs markedly from traditional definitions, which are often concerned exclusively with response time, most usually defined in absolute terms rather than in human terms. Thus, an operating system like RT-11 measures responsiveness in terms that are important for real-time events such as data collection from satellites or process control monitoring, but has little direct support for the type of programs that run on a personal computer.

The contradiction between the requirements of richness and responsiveness creates something like a cycle of reincarnation. Systems increase in richness to the point where their responsiveness becomes unsatisfactory. At that point, improvements in responsiveness are undertaken, even at the cost of decreasing richness, to the point where richness can again begin to increase. Precisely this phenomena was observed in the design of graphics display processors, the precursors of today's personal computers [8].

We are interested in defining the levels of performance at which the temporal, spatial, and chromatic properties of computational environments are satisfactory for human users. Our primary research interest is in studying responsiveness. To accomplish this we are conducting specific experiments, with the help of cognitive psychologists, to determine the parameters of real-time human performance. Our work has included investigations in the spatial domain [3], the chromatic domain [9,11], and interaction techniques for specifying curves [4]. We have in fact been forced to face the question of the trade-offs between richness and responsibility head-on in our design of the support system for our experiments. The dilemma we encountered can be seen in the following two descriptions of our efforts.

The part of the system in which the design of an experiment occurs needs richness: graphical design tools for manipulating multi-task models of experiments, high-level real-time multi-task debugging, real-time simulation tools, on-line monitoring of

experimentation, proofs that timing constraints can be met, and so on. The experiment designers need these elaborate tools to ensure that the design meets the most exacting standards of experimental methodology; the designer can forego the luxury of perfect responsiveness (although reluctantly).

An almost disjoint part of the system, which runs only when the experiments are being conducted, is required to have a high enough level of responsiveness such that no artifacts are perceptible to the subject. In contrast to the designer's tools, the subject's view of a psychology experiment is very impoverished; the choice of possible interactions is deliberately limited by the experimenter. Here the trade-off is made the other way. Richness is readily sacrificed for responsiveness.

We have found that there is an advantage in splitting richness and responsivity apart into different aspects of our system where they can be dealt with in isolation. A pleasant benefit of this decision is the recursive feedback loop it provides. The system is used for interactively designing experiments; the results of the experiments help us to set standards for responsiveness in interactive systems; the knowledge gained can be applied to the design phase of the system to improve its responsiveness as a rich set of tools.

3. The Experiment Designer's Desiderata

When considering experiment specification, we can look at two more or less separate views. The view that the experimental subject sees is simpler to describe, so we will treat it first, leaving until later the view that the experiment designer sees.

The system as seen by the experimental subject has two logical components: screens and input devices. Screens present images, which have temporal, spatial, and chromatic properties created by the experimental designer. Ideally a workstation will neither add nor subtract perceivable information from the images it presents. This criterion places rather stringent performance requirements on the screens and screen drivers, which are inherently digital devices presenting analog information. The next three examples discuss representative parameter limits for different aspects of screen design.

In the temporal domain, the visual system can detect modulation up to 70 Hz under optimal circumstances. Thus, a refresh rate of at least 70 Hz is needed to present artifact-free stationary stimuli. Some commercial displays are beginning to meet this specification (the Macintosh uses a refresh rate of 67 Hz). For temporally modulated or moving stimuli, a refresh rate twice 70 Hz is needed [5]. Open questions remain: "to what extent can temporal antialiasing mask otherwise visible artifacts?" and "why don't people complain when watching motion on broadcast television or at the movies?"

Spatially there exist two limits: a *normal acuity limit* at about 1 minute of visual angle and a *vernier acuity limit* at about 6 seconds of visual angle. Screen size should cover at least 30° to 60° for any viewing experiments and pixels should be small enough to produce line widths and position uncertainties below what is visible, so that aliasing artifacts do not change the image appearance. To put this in terms of current displays, a 1000×1000 screen with pixel size adjusted to the normal acuity limit subtends 16° of visual angle, a little below the minimum viewing angle. But a 1000×1000 screen with pixel size adjusted to the vernier acuity limit subtends only 1.6° of visual angle, more than a factor of ten below the minimum acceptable viewing angle [12].

Techniques such as antialiasing lower the screen resolution needed to faithfully reproduce a given image, but work to date testing these techniques in a rigorously controlled visual environment [3] has only scratched the surface. There remain difficult experimental questions, such as "for what purposes is the color of an antialiased letter 'the same' as the color of a equal-sized letter presented non-antialiased on a higher resolution display?" that must be examined.

In the chromatic domain, the size of visible artifacts depends on the rate of variation of color on the screen. When color is varying rapidly (near the limit of spatial acuity) artifacts over 10% can be invisible. But when color is varying very slowly, artifacts of as little as 0.1% are near the threshold of visibility. Measurements done by Kelly [7] provide a rough framework in which the trade-offs can be evaluated, but these need to be extended. Methods for calculating other trade-offs between color quantization and dynamic range appear in the literature [5]. There has been, to our knowledge, no attempt to take advantage of this trade-off in screen design, either in hardware or software.

On the input side, spatial factors seem to be better-understood. Using natural non-linearity in mouse motion to remove hand tremor and to speed up large movements, for example, has adequate engineering solutions, although there is not a satisfactory motor theory to explain why these methods work. Timing is more difficult. Many informal reports make it clear that there is a value of lag between the input and the screen update at which the user switches from an "adjusting the screen" metaphor to an "adjusting something that adjusts the screen" metaphor, with a large drop in performance. Experimental data providing sound values for this parameter are not available. However, it is possible to put an upper limit on performance: there is no need to update internal values faster than the refresh rate of the screen (although temporal antialiasing may create a logical refresh rate that is above the physical refresh rate).

To see what tools the experiment designer needs to create a software configuration that meets the constraints, consider the logical structure of an experiment implemented on a workstation. The software complement of the experiment is a collection of tasks that can be combined in different ways to

produce a variety of experiments or interactions. The designer should present an experiment specification, which is a program in an interpreted *experiment control language*. Software on the workstation should interpret the specification, dynamically configuring and reconfiguring tasks to produce the experiment. The metaphor used by the experimental designer is that of a collection of servants that, when given the appropriate orders, carry out the many atomic tasks that comprise the experiment. Thus the experimental designer requires three types of functionality: specification creation aids, experiment monitoring aids, and specification debugging aids.

Specification Creation Aids

Our model of an experimental session is that the configuration at any given moment depends on session-dependent information that remains constant throughout the session and on trial dependent information that varies from trial to trial within a session. We expect the experimental designer to create session and trial prototypes, then to specify sets of values for variable parameters in the prototypes allowing a specification file containing appropriate combinations of the values. The designer must be able to arrange the atomic tasks so that all the components of the experimental session obey whatever timing constraints he wishes to impose. We envision a graphical interface where the geometric arrangement of the tasks on the screen indicates the dependence and communication relations that define the experiment and which, upon completion of the interactive session, will be translated into an experiment specification file.

Other requirements for experiment creation are tied closely to attributes of the experimental run-time environment. For example, the experiment designer should be able to run a sample trial of the current configuration. Part of this capability should be running the trial at a variety of slowed down rates, using "canned" inputs, or allowing the designer to contribute inputs the subject would have to make during the running experiment. To connect tasks together for communication purposes we need to have a good definition of possible message semantics between the tasks, and the more parsimonious the semantics the better, because the ideal is to be able to connect any task to any other task. It must be possible to insert timing probes and controls into the atomic tasks with minimal alteration of their structure.

Specification Monitoring Aids

It is not sufficient to monitor an experiment simply by watching its progress and checking for artifacts or anomalies. The designer must either employ proof-of-correctness techniques to show that the experiment specification will produce results within the desired range of parameter values, or the designer must monitor each running of the specification to give a retrospective range of parameter values for each run.

The first is more desirable, but less likely to be feasible in actual cases. The second can be included as a "built-in feature" working from requirements given by the experiment designer to produce automatically a record of the values of parameters and an alert if they ever fail to fall in the allowable range of values. At best this serves as an operational proof of correctness when the experiment specification is run on a well-selected set of conditions and adjusted not to fail. When failures do occur, it serves as a back-up, allowing us to discard the small number of trials or sessions where the range of parameter values is not maintained. At worst it functions to indicate only what trials to discard. But if the discarded proportion of trials is too high, the experiment must be abandoned, because the monitoring process gives no method for altering the specification.

Specification Debugging Aids

Inevitably, there will be many cases where a configuration created by an experimental designer fails to produce the trial he desired. He then needs a debugging tool that allows him to visualize the operation of the trial at a high level and associate parts of the specification with corresponding parts of the trial. Detecting syntactic errors such as when a message output is connected to an incompatible message input should be part of the interactive configuration controller. Semantic errors, in contrast, come in so many flavors that a single tool is unlikely to provide a complete debugging capability. At one extreme, errors in configuration logic can be illuminated by showing activity in the atomic tasks and in their communication channels simultaneously with the interpreted trial.

At the other extreme, failures to meet timing constraints require the ability to run the trial simulating full speed with the trial monitor engaged. The debugger should then be able to isolate the synchronization actions associated with the violation of a particular constraint, and should be able to perform something like a critical path analysis to isolate the parts of the communication flow that cause the constraint violation.

Between these two solvable extremes lies the type of case which is likely to be most common: where the experimental designer simply omits one or more constraints that are necessary for the trials to meet his requirements. How should the configuring program be set up to demand enough constraints to completely disambiguate the specification? How should the configuring program communicate its default behaviors to the experimental designer? To what extent can a debugger suggest plausible constraints to the experimental designer? All these questions, and many more like them, lead straight to the heart of difficult and fundamental problems in human-computer interaction.

4. The Waterloo Experiment Manager

The Waterloo Experiment Manager implements many of the ideas expressed in the Desiderata. It is composed of two distinct processing systems, the

experiment design environment and the *experiment controller*. The experiment design environment is the rich environment for specifying and analyzing experiments concerning human factors in interactive graphics. The experiment controller is the responsive environment in which the experiments can be run without fear of compromise by the introduction of perceptible artifacts in the temporal, spatial, or chromatic domains.

The Experiment Design Environment

The current implementation of the experiment design environment is a production program capable of making up experiment specifications given a set of pre-programmed tasks configured on the experiment controller. It works at the level of direct manipulation of an *experiment specification file*, so its capabilities are primitive compared to those espoused in the Desiderata. Nonetheless, it is versatile enough to have seen daily use for about two years at the National Research Council of Canada generating specification files for sequential experiments in a single process operating system.

The experiment design environment uses prototypes of the session parameter and trial parameter parts of the specification file. Values for parameters in the prototypes can be taken from archive-style dictionaries or can be introduced interactively. They can be selected randomly from the dictionaries, with or without replacement, or can be used exhaustively to create crossed designs. Trials can be separated into blocks and randomized within blocks. At this level of routine experiment management we are continuing to improve the tool: adding partially balanced incomplete block designs, incomplete designs like Latin squares, constrained randomization, and so on. These capabilities must be built into the lower levels of any tool that will create specification files.

In the context of this tool, we are considering how to incorporate syntactic cross-checking between the prototypes on which the specification program is based and the input modules of the atomic tasks that comprise the experiment. Use of this information will remove the chief source of error in the current version of the experiment design environment.

The Experiment Controller

The experiment controller runs the experiment defined by the experiment specification file. All information presented to the subject by way of the graphics displays or other output devices and all input from the subject through the interactive devices are handled by the controller. Comprised of several M68000 series processors, the experiment controller is based on the Harmony multiprocessor multitasking operating system [6]. The use of a real-time operating system has contributed to the design of the system in two ways, ensuring responsive behavior and providing a multitasking metaphor on which the building block approach to experiment design is based [1,2].

A real-time, multitasking, multiprocessing system ensures responsive behavior in several ways. Time critical operations are performed by high-priority tasks without fear of the operating system stealing time for its own use. Processing may be distributed over several processors to avoid conflict between these priority tasks. Access to peripheral devices is direct, requiring little or no operating system intervention that would slow down the process.

The multitasking approach used in Harmony (and other related systems) is based on the idea of building systems using a large number of small tasks with specific duties. Using inexpensive *synchronous message passing* through *send-receive-reply* primitives for inter-task communication and rapid task creation and destruction operations, systems are quickly and easily configurable and reconfigurable.

The experiment controller tasks fall into two categories, *daemon tasks* and *generic tasks*. Daemons are the building blocks from which experiments are constructed. For example, the color matching experiment described in the appendix requires a daemon that creates a window on the screen, one that gets input from the graphics tablet, and one that uses the locator input to control the color displayed in a window. These daemons may be connected to a data logging daemon so that events noted by the color changing daemon are stored for post-experiment analysis.

Generic tasks are automatically created for each experiment. Two examples are the *experiment runner*, which performs the initial processing of the experiment specification file, and the *connection administrator* or *patch panel*, which supervises the communication between daemons.

The Experiment Specification File

The experiment specification file controls the experiment. Passed from the design environment to the experiment runner, the file is composed of *blocks*, each of which contains a series of commands. These commands refer to the daemons that exist in the experiment controller. Each daemon is a task for which compiled code has been down-loaded prior to the experiment.

The first block in the experiment specification file is a *daemon block*. It specifies the daemons that will be required for the experiment. This is the tool by which the experiment designer specifies the configuration of the experiment. If a daemon is required to exist more that once, for example a filter that is needed in two communication paths within the experiment, that daemon may be requested more than once and multiple instances of it will be created.

Following the daemon block is a *session block* and a sequence of *trial blocks*. The format of these two types of blocks is the same. Each contains a sequence of commands to be executed during the experiment. The session block is performed once at the start of the session, followed by each of the trial blocks in turn.

The commands in the session and trial blocks are in a textual format. For example, the command

```
Image_poster ( cmd = RECTANGLE, scr_name = S1,
        wind_name = Wind_a, ll_x = 150, ll_y = 400,
        ur_x = 350, ur_y = 600, color = 1, red = 350,
        green = 850, blue = 593 )
```

is forwarded to the daemon `Image_poster` and tells it to create a rectangle on screen one (the system may have several frame buffers controlling several displays) in a specified window. It also gives the lower left and upper right coordinates of the rectangle, and the color look-up-table entry to use.

The experiment runner sends the commands to the daemons in the form of a reply to their requests. In this way, the runner remains unblocked, and can forward commands to several daemons so that they will be active simultaneously. However, there must be some technique for synchronization – some way to hold back operations until an event (temporal, input, or other) occurs. The synchronization tool is the `Wait for` statement of the experiment specification file. These are interpreted by the experiment runner and are the only mechanism by which it can be blocked. Each of these statements contains a Boolean expression of daemon names, which causes the experiment runner to wait until those daemons have processed their commands and have sent requests for more work. For example,

```
Wait for ( Clock or Termination )
```

causes the experiment runner to stop processing the experiment definition file until either the `Clock` daemon (which will not return for more work until a specified amount of time has elapsed) or the `Termination` daemon (which returns only when certain conditions occur that indicate the end of a trial) sends a message to the experiment runner.

The case illustrated above, in which there is a Boolean "or" function in the `Wait for` statement, illustrates a potential problem with the daemon approach. At the end of the trial, one or several daemons may be waiting for a condition that may not occur for some time, if ever. Having not finished, they cannot be forwarded the command for the next trial. The experiment runner handles this situation by simply destroying the lost daemons and creating replacements. Thus the controller provides a mechanism where any task in the system may be removed and replaced – either as a reset for the next trial, or after a timeout condition. This contributes to the configurability of the system and permits the addition of timing controls without any change to the operations that are being timed.

Daemons may be general, in that they are used in a number of experiments, or specific, being written for a single experiment then thrown away. Our aim is to write configurable daemons so that few new daemons will have to be programmed for any specific experiment. The method by which a daemon is informed of its configuration parameters is through the daemon commands that follow the daemon block. Session parameters will be sent to the daemon at the beginning of the session, trial parameters will be sent at the beginning of each trial.

Inter-Task Communication

An experiment is defined not only by the activities that take place in each trial, as embodied in the trial daemons, but also by the interaction between those activities. A trial can be seen as consisting of one or more *information pipelines*. Input from the subject passes through the pipeline undergoing various transformations to eventually be reflected in the desired output form(s). These transformations are effected by the daemons that are the components of the experiment, implemented as Harmony tasks.

The interaction between the various trial daemons can be viewed as a system of relationships between tasks that produce information and tasks that consume information. A task can be both a consumer and a producer of information. Every active daemon is connected to at least one other task in the experiment. The *producer-consumer model* of the system's behavior comes naturally from the need to provide the designer with a simple conceptual model of the available environment, one that is configurable and extensible.

Understanding the behavior of the system in terms of these communication channels encourages the building block approach. Daemons can be assembled into an information pipeline in a variety of ways. This discourages the traditional tendency (often imposed by a programmer-designed system) to divide tasks into input and output "camps", wherein a task is seen as either interpreting information obtained from an input device or transforming information destined for an output device. Input in the producer-consumer model is received from a producer, whether a device, another trial daemon, or a system task. It is forwarded to the specified consumer, without any restrictions being placed on the source or destination.

We emphasize that the producer-consumer model describes the experiment system in terms of organization rather than in terms of the characteristics of the particular trial daemons. This cleanly separates the concept of a daemon's *functionality* – what it does with the information it receives (i.e., take in a 16-bit integer and use it to transform an entry in a color lookup table) – from the concept of its *dependencies* within the overall system (whence it receives its input and to where it directs its output). This makes the writing of future tools, whether trial daemons or system tasks, very simple; the programmer need only be concerned about the particular message semantics.

The Patch Panel

A major goal of the project is to provide the designer flexibility in organizing the tools at his disposal. Flexibility is supported by a system task, the *patch panel*, which is based on the model of the system as a collection of producer and consumer tasks [1]. With roots in the switchboard model used in the Adagio system [10], the patch panel provides a means of

dynamically connecting the output channel of a producer to the input channel of a consumer, much as a physical patch panel is used in electronics to permit easy routing of signals from one module to another. In the Harmony system model, a communication channel between two tasks can be considered a system resource, and the patch panel a *connection manager* responsible for allocating and administering those resources.

In keeping with the design of the experiment controller, the patch panel is created by the experiment runner and is passed a list of producer-consumer *connection requests*. Each connection request is accompanied by certain parameters specifying the details of that information flow. The patch panel sets up the connections and retains the information relevant to each.

While the patch panel reflects the overall information flow organization of a particular experiment, it contains no semantics related to particular tasks, nor to the information passing between tasks. All syntactic and semantic checking of messages between producer and consumer is left to the tasks themselves; the patch panel only knows about the routing and buffering details of the communication channels that carry messages.

Isolating the implementation and administration of the connections in the patch panel abstraction is advantageous for several reasons. It facilitates the specification of information flow; the designer merely sets a list of producer-consumer connections and the parameters of each connection. This again supports the building block approach, wherein tasks can be combined in a variety of ways, with these combinations constrained only by compatibility in the type of information exchanged between producer and consumer.

Localizing the responsibility for implementing connections within the patch panel provides a useful metaphor of the system for the designer, one that is both conceptually simple and yet powerful enough to describe the experiment's organization.

The patch panel enables a many-to-many mapping between producers and consumers. A task can receive input from an arbitrary set of producers and produce output for an arbitrary set of consumers. This is essential when the designer wants a task to take its input from more than one device or have more than one task driven from the same device.

5. Conclusions

We have implemented an architecture for conducting experiments concerning human factors of interactive graphics consisting of host-based design tools using a conventional Unix or Unix-like mainframe and an embedded microprocessor-based workstation. The division of labor between the two subsystems is determined by the requirements for a rich design environment and a responsive run-time environment.

The current implementation includes a relatively complete set of tasks in the experiment controller and a very general protocol for communicating the experimental design to the experiment controller. The distinction between generic and specific tasks in the experiment controller is an important one. By requiring each daemon task to accept configuration parameters, the experiment controller becomes an extensible tool that needs relatively little modification when new experiments are designed.

A version of the experiment design environment is operational. Extension to a much richer set of tools is in progress. The experiment specification files currently used are a combination of some generated automatically from higher-level specifications and some generated manually. Subsequent versions of the experiment design environment will provide significantly more automation at a much higher level in the design process.

The methodology advocated here applies to more than just experiments in cognitive psychology. We believe that the same approach can be applied to the design and implementation of a wide variety of interactive graphics systems, including bank teller machines, transaction processing terminals, and engineering workstation. These all provide interactive computing to a user. As we move along the continuum, the task domain becomes more complicated, the facilities offered become richer, the user population for a particular machine becomes smaller, and the length of interaction with a single user becomes longer. But the concepts developed in this paper apply to the whole continuum and the lessons learned about the experimental designer's task are relevant to all of the systems.

References

[1] L. R. Bartram, and P. P. Tanner, "Configurable Multitasking: The Building Block Approach to Interactive System Design," *Submitted to IEEE Software*, (January 1989).

[2] K. S. Booth, W. B. Cowan and D. R. Forsey, "Multitasking Support in a Graphics Workstation," *Proc. 1st International Conference on Computer Workstations*, (November, 1985) pp. 82-89.

[3] K. S. Booth, M. P. Bryden, W. B. Cowan, M. F. Morgan and B. L. Plante, "On the Parameters of Human Visual Performance, An Investigation of the Benefits of Antialiasing," *Proceedings of CHI+GI 1987*, (April, 1987) pp. 13-19.

[4] E. G. Bosch, R. H. Bartels, K. S. Booth and P. Jolicoeur, "Workstation-Based Shape Matching Experiments," *2nd IEEE Conference on Computer Workstations*, (March, 1988) pp. 132-141.

[5] W. B. Cowan, "Discreteness Artifacts in Raster Display Systems," *Colour Vision: Psychophysics and Physiology*, pp. 145-154, Academic Press: London, 1983.

[6] W. M. Gentleman, "Using the Harmony Operating System," *Tech. report NRCC-ERB-966* Division of Electrical Engineering, National Research Council of Canada, December, 1983.

[7] D. H. Kelly, "Visual Responses to Time Dependent Stimuli I. Amplitude Sensitivity Measurements," *Journal of the Optical Society of America*, vol 51, pp. 422-429, 1961.

[8] T. H. Myer and I. E. Sutherland, "On the Design of Display Processors," *Communications of ACM*, vol 11(6), pp. 410-414, June 1968.

[9] M. W. Schwartz, W. B. Cowan and J. C. Beatty, "An Experimental Comparison of RGB, YIQ, LAB, HSV and Opponent Color Models," *ACM Transactions on Graphics*, vol 6(2), pp. 123-158, April 1987.

[10] P. P. Tanner, S. A. MacKay, D. A. Stewart, and M. Wein, "A Multitasking Switchboard Approach to User Interface Management," Proc. SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986) *Computer Graphics*, vol 20(4) pp. 241-248.

[11] C. Ware and J. C. Beatty, "Using Colour to Display Structures in Multidimensional Discrete Data," *Color Research and Application 11* (June, 1986) pp. S11-S14.

[12] G. Westheimer and S. P. McKee, "Visual Acuity in the Presence of Retinal Image Motion," *Journal of the Optical Society of America*, vol 65, pp. 847-850, 1975.

Appendix

A Color-Matching Experiment

To illustrate the activities of the experiment controller, we can look at a simplified experiment specification file and the resulting activities of the tasks within the experiment controller. The experiment specification file on the next two pages controls a color matching experiment [9]. In the experiment, two rectangles are placed on the screen for each trial, one with a target color, the other with a manipulable color. The subject is required to use a graphics tablet to control the color of the manipulable rectangle until it matches, as well as possible, the target color.

The daemon block of commands in the file specifies the seven tasks that must be created for the experiment. These are standard tasks that can be used for a variety of experiments investigating the interactive manipulation of color.

The session block sets up the experiment, initializing the screen, claiming screen real estate for the experiment, initializing the clock and the tablet, and initializing two color-changing daemons (instances of the same task). These two daemons will take input from the tablet and change the color of the manipulable color according to the input.

Two trial blocks are shown. A full experiment usually has many trials in a session. In each trial, the two rectangles are drawn simultaneously. A clock daemon is set up to report back in 40 seconds, and a termination condition daemon will report back when a tablet button is pushed to indicate subject satisfaction with the color match. This provides a time-out mechanism to abort a trial if the subject takes too long to perform a match. The trial will end when the first of these events occurs.

Figure 1 indicates the flow of the commands through the experiment runner to the daemons. Daemons receive commands both during the session initialization (indicated with solid lines) and during each trial. Figure 2 shows the flow of information from information-producing daemons to information-consuming daemons during the actual trial.

An Experiment Specification File: The Daemon and Session Blocks

```
/* Specify required daemons for session */
daemons
{
  Clock           ( dtype=CLOCK );
  Color_changer1  ( dtype=COLOR_CHANGER );
  Color_changer2  ( dtype=COLOR_CHANGER );
  Image_poster    ( dtype=IMAGE_POSTER );
  Screen_manager  ( dtype=SCREEN_MANAGER );
  Tablet          ( dtype=TABLET );
  Termination     ( dtype=TERMINATION );;
}

/* Session initialization instructions */
session
{
  Screen_manager( cmd = INIT_SCREEN, board_num = 1,
          scr_name = S1 );
  Screen_manager( cmd = NEW_WIND, scr_name = S1, ll_x = 0,
          ll_y = 0, ur_x = 1000, ur_y = 1000,
          wind_name = Wind_a );

  Wait for Screen_manager ;;

  Clock( cmd = INIT_CLOCK );
  Wait for Clock;;

  Tablet( cmd = INIT_TABLET, tab_name = Tab_output );
  Tablet( cmd = START_TABLET );

/* Don't wait for tablet; START_TABLET is an infinite
 * loop
 */

  Color_changer1( cmd=CONNECT, scr_name=S1, color=2,
          input=Tab_output, axis = X, gun = Red );

  Color_changer2( cmd=CONNECT, scr_name=S1, color=2,
          input=Tab_output, axis = Y, gun = Blue );;
}
```

Experiment Specification File: The Trial Blocks

```
trial
{
/*
* Draw two squares of different colors, and allow the
* user to change the amounts of red and blue in one of
* them Make sure they both have the same amount of green
* so a perfect match is possible.
*/

  Image_poster( cmd = RECTANGLE, scr_name = S1,
          wind_name = Wind_a, ll_x = 150, ll_y = 400,
          ur_x = 350, ur_y = 600, color = 1, red = 350,
          green = 850, blue = 593 );

  Image_poster( cmd = RECTANGLE, scr_name = S1,
          wind_name = Wind_a, ll_x = 650, ll_y = 400,
          ur_x = 850, ur_y = 600, color = 2, red = 0,
          green = 850, blue = 0 );

  /* Use the clock to get a timeout of 40 seconds */

  Clock( cmd = DELAY, millisec = 40000 );
  Termination( cmd = ANY_BUTTON_CHANGE, tab_name=Tab_out );

  Wait for( Clock or Termination );;

  /* Trial is over; clear screen */

  Screen_manager( cmd = CLEAR_SCREEN, scr_name = S1 );;
}

trial
{
  Image_poster( cmd = RECTANGLE, scr_name = S1,
          wind_name = Wind_a, ll_x = 150, ll_y = 400,
          ur_x = 350, ur_y = 600, color = 1, red = 900,
          green = 850, blue = 200 );

  Image_poster( cmd = RECTANGLE, scr_name = S1,
          wind_name = Wind_a, ll_x = 650, ll_y = 400,
          ur_x = 850, ur_y = 600, color = 2, red = 0,
          green = 850, blue = 0 );

  Clock( cmd = DELAY, millisec = 40000 );
  Termination( cmd = ANY_BUTTON_CHANGE, tab_name=Tab_out );

  Wait for( Clock or Termination );;

  /* Trial is over; clear screen */

  Screen_manager( cmd = CLEAR_SCREEN, scr_name = S1 );;
}
```

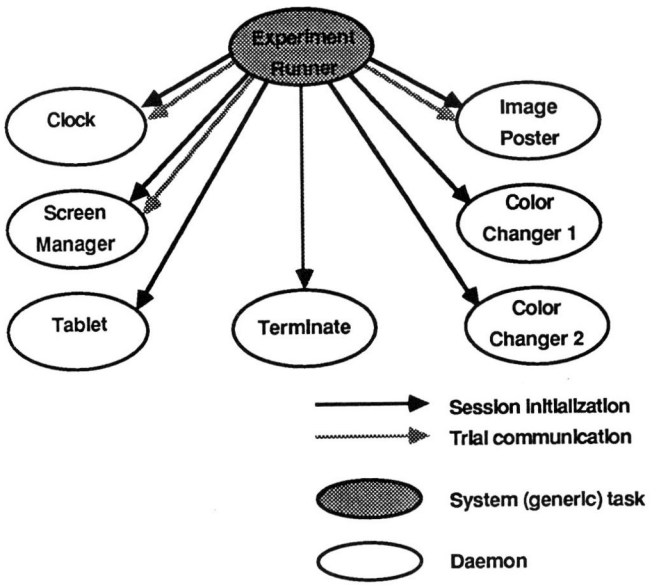

Figure 1. The flow of daemon commands at initialization.

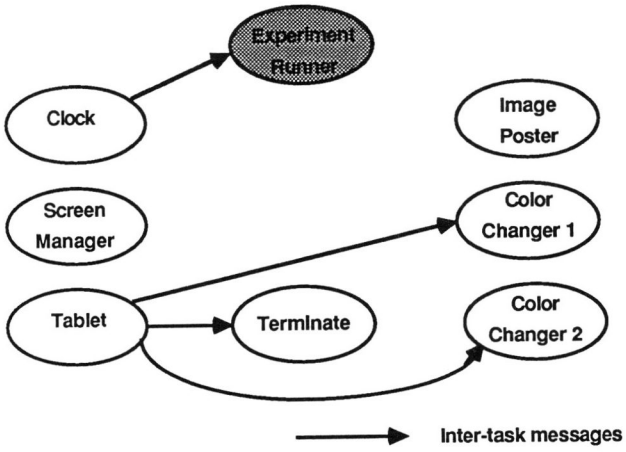

Figure 2. The producer-consumer relationship among daemons.