In this example, the digital filtering operation has three stages. Each image of the movie is filtered spatially (each pixel is processed in terms of surrounding pixels) and temporally (each pixel is processed in terms of corresponding pixels in previous and following images). A threshold operation then zeroes pixels below a given value.

An interactive filtering application is created through the use of a number of communicating modules (Figure 5-1). Spatial and Temporal filter modules are created independently, each having input and output image published variables. However, the spatial filtering operation is computationally intensive. For this reason, a Dispatch module is used to distribute the images between two identical Spatial modules running concurrently on separate processors. The spatially filtered images are merged at the Temporal module, filtered, and sent to the Threshold module. EUPHORIA is used to control the filtering operation; users specify the movie name and threshold value. The original and filtered movies are also displayed within EUPHORIA.

Obviously, the modules of this application should run concurrently on separate processors. Running all of the modules on the same processor would severely hinder performance since none of the modules would run in parallel. Each module has different performance characteristics that should be considered in deciding how to allocate modules to available processors. The Spatial module is the most computationally intensive, therefore its two instantiations should be run on the
fastest two processors with the lightest loads. The Server and Dispatch modules just forward image information among modules, and therefore do not need to execute on fast processors. However, the network connections among these modules should be fast (e.g., running the Server on a workstation in Japan and the Dispatch on a workstation in Brazil would result in slow performance). The EUPHORIA module is usually fastest running on the user's local workstation due to fewer communication delays among the X Windows client and server portions [66]. All of these factors should be considered when launching this application.

5.2.2 Vaudeville Video Teleconferencing Application

Vaudeville is an ATM-based teleconferencing application that supports scalable, multi-party video teleconferences [57]. Each conference participant uses special hardware called the MultiMedia eXplorer (MMX) [60], along with an associated video camera and microphone, to send, receive and view NTSC-quality live video. In addition, each participant has a graphical user interface (Figure 5-2) that displays still images of all conference participants, the currently selected speaker, and various controls (e.g., volume, mute).

![Figure 5-2: Vaudeville Video Teleconferencing GUI.](image)

Vaudeville is an example of a client/server application (Figure 5-3). The server portion consists of modules maintaining information about the conference participants and the currently selected
speaker. The client portion consists of modules that are specific to an individual participant. When a user wants to join a conference, it is necessary to launch and configure the client modules to the appropriate server modules. Each client module has different performance requirements. For example, each client has an MMX module that must run on the user’s local machine in order to access the MMX hardware. Clearly, it is undesirable to expose the end-user to these details. Instead, a mechanism to automatically launch and configure the client to an end-user selected conference would be preferable.

![Image](image_url)

Figure 5-3: Joining a new participant to a conference.

Vaudeville was developed prior to the application management system described in this chapter. Launching and configuration is hard-coded into Vaudeville’s implementation, therefore, end-users are currently not exposed to these details. Application management provides a mechanism to automate this process, simplifying the implementation and making the application more flexible.

### 5.3 Related Work

The World Wide Web (the Web) [2] provides Internet users with the ability to share a variety of information, viewable with the use of a graphical browser. Although sharing information over the Internet was possible for many years prior to the creation of the Web (e.g., ftp, gopher), giving end-users the ability to use this information has resulted in an unprecedented increase in Internet use. The hypertext markup language (HTML) provided users with the ability to create their own web pages, requiring little technical expertise. The philosophy of this dissertation is similar to the philosophy of the Web; just as the Web provides end-users with the ability to create hypertext documents, we want to provide end-users with the ability to create distributed multimedia applications and make the applications available to others over the Internet.
Java [1], [11] is an object-oriented programming language developed for use with the World Wide Web. Java programs can be included in a web page, allowing application developers to make their applications available to other Web users. A Java program is compiled as machine independent bytecode that can be rapidly downloaded over the Internet from where it is stored. The bytecode is interpreted by the target machine at run-time. This approach seems well suited to small applications imbedded within a Web page (called "applets"), but it remains to be seen whether more substantial applications can be developed in this way.

Due to the vastness of the Internet and the Web, many name and location services have been developed. Services on the Web include the TCS Virtual Rolodex¹, the World Wide Web Virtual Library², and the World Wide Web Yellow Pages³. The DCE Web [36] applies a set of distributed computing technologies, based on OSF DCE, to solving the authorization and organizational problems of the Web. The DCE Web also incorporates a name-service based mechanism for locating objects.

OpenUI [55] is a user interface management system designed specifically for client/server and three-tiered enterprise distributed applications. OpenUI applications consist of a graphical front-end, application logic, and a distributed database management system. The OpenUI development environment includes a graphics toolkit, a messaging architecture, a visual builder for constructing “dialog boxes,” and OpenWeb, a mechanism for launching user interfaces for OpenUI. To make a client/server application available to users on the Web, developers create a link on their Web page that points to the downloadable user interface specification. Web users who have installed the OpenUI “helper application” can then download user interface specification, using the helper application as the graphical front-end of the (already running) application. The application management functionality described in this dissertation is much more general, allowing distributed applications consisting of an arbitrary number of components to be launched and dynamically configured.

¹. http://hercule.csci.unt.edu:80/tcs-rolodex/
². http://www.w3.org/hypertext/DataSources/bySubject/Overview.html
A number of techniques have been tried to deal with distributed systems and large resource bases. PVM [71] and PRM [52] both address the distributed access problem by employing a hierarchy of control agents to allocate jobs to processors. The application management mechanism described within this document is similar to PRM. Like PRM, we have chosen a hierarchical model of resource control. PRM has a hierarchy of different types “managers” that are used to launch the components of a distributed application; these managers are similar to the brokerage system described here. The main differences between our approach and theirs is in the user interface, application specification, and configuration. PRM does not provide a mechanism that allows end-users to launch applications; we provide an intuitive World Wide Web mechanism. In this way, our applications can be easily distributed to the entire Internet community. In PRM, the “job manager” is used to decide how to allocate processes to machines. The way of specializing how the components of an application are allocated is to customize the job manager (i.e., through programming). Our method of specifying the application is generic (i.e., no programming), allowing the application designer to state preferences about each module’s desired performance that the system will attempt to enforce. Finally, PRM has no built-in configuration mechanism; it only allocates jobs to machines. Our system performs the configuration among the modules of a Playground application. In addition, it also supports a specialized configuration mechanism for client/server applications, connecting client modules to server modules. No programming is required to specify this communication structure.

5.4 Using a Distributed Multimedia Application

End-users interact with the application management mechanism through the use of application pages. Each application page is a World Wide Web document that is viewable through the use of a Java-enabled Web browser (e.g., Netscape Navigator [51]). Figure 5-4 contains an example application page for the Vaudeville Video Teleconferencing application (Section 5.2.2).

An application page contains all relevant application documentation. In addition, the page also has a table of running application instantiations and several buttons for using the application and getting more information. The GRAB button imports running modules of a selected application into a local visual configuration tool. The NEW button launches and configures the modules of the application. The JOIN button is used to launch the modules of a “client” portion of an application
and configure the communication to the already running modules of the "server" application. With NEW, the end-user is prompted for some brief documentation information and other options (e.g., register application with page). With GRAB and JOIN, the end-user must first select an item from the running application list. The chosen option and its associated information is communicated to the end-user's brokerage system (see below).

Application pages could be arranged to facilitate the location of components based on specifications. For example, one could create a page that has links to application pages for filtering an image. In addition, keywords describing an application's functionality could be included within the application page, enabling end-users to find components through the use of a Web search tool.
5.5 Task Outline

Launching a complete distributed application requires many steps and a flexible control structure. Ideally, the control system would scale well with increasing available resources. It should be flexible enough to allow local management of resources, yet still follow global requirements set down by the application designer. The end goal of the control system is to allocate the available resources to provide the fastest response and throughput for the application.

For example, launching and configuring the Medical Image Processing Pipeline application (Section 5.2.1) involves the following tasks. First, the list of required modules and their configuration is obtained, perhaps remotely from an external file system. Second, the current performance information about each available processor, including the load and network traffic, is gathered and evaluated. Third, decisions are made on how to allocate modules to processes, considering the configuration, performance requirements, and current processor information. Fourth, the executables for the modules are located on the processors' file systems and are launched on the appropriate processors. Various types failures (e.g., a workstation is rebooted) are handled appropriately. Fifth, the communication structure (i.e., logical connections) is established according to the configuration. Finally, running module information is stored somewhere to facilitate application shutdown and to enable others to join the application.

5.6 Design

Given the end-user selection of how to use an application, a brokerage system is used to launch and configure applications. Figure 5-5 illustrates the relationships among the application management system components. Each application page has an associated application directory that contains all information specific to the application. This includes the list of modules comprising the application, the configuration among the application modules, and information about each running application instantiation. The application directory is created by the application designer and resides as a world-readable directory on the designer's own file system.

Join information is stored in the form of a parameterized configuration. With the I/O Abstraction model, application modules are written independent of their configuration. Parameterized
configurations allow the application designer to specify client/server "join" operations without doing any textual programming, exploiting I/O Abstraction's separation of communication from computation.

Each application directory is managed by an application daemon module that has the appropriate permissions to write files to the application directory. The purpose of the application daemon is simplify access to the application directory, allowing modules (possibly running on different file systems) to read and write application information.

![Diagram of the Application Launch architecture](image)

Figure 5-5: The Application Launch architecture.

When an application page is viewed with a Java-enabled Web browser, a Java program called the liaison applet is launched. The liaison applet communicates with another Java process called the liaison server, which periodically scans the application directory, keeping a list of running application instantiations (Figure 5-4). This list is communicated to the liaison applet which displays it in a graphical table. The main reason for a separate applet and server was to avoid protection issues associated with applets and to make the applet portion as small as possible (i.e., to minimize the amount of time that the user has to wait in loading the applet in their browser).
After the end-user has selected the appropriate option in the application page, the liaison server communicates the run option to the end-user’s brokerage system.

The purpose of the brokerage system is to launch and configure application modules. This system consists of a broker module and one or more hierarchies of launcher modules. Each end-user runs their own personal copy of the broker. The application daemon forms connections to the end-user’s broker in order to communicate application information to the broker and to receive information about newly launched applications. The specification of each module and the configuration is read by the application daemon from the application directory and is communicated to the broker. The broker delegates the task of module launching to its launchers, gathers module IDs of newly launched modules, configures the communication among modules, and communicates the set of IDs back to the application daemon.

The task of the launcher is to launch a collection of modules, optimizing module allocation to processors according to some heuristic based on module performance requirements, hardware configurations, and other factors. Each launcher controls access to some portion of the total computing resource, including other launchers and physical CPUs. Launchers may form a hierarchy, as seen in Figure 5-5. "Leaf" launchers gather information about a single workstation (e.g., its current load, list of supported modules) launching modules on their own workstation. Other launchers manage a collection of child launchers, summarizing child information to parents and delegating the launching task to children. A launcher may conceal the details of resources it controls, acting as a firewall to those resources.

5.7 Summary

This chapter described an application management system, enabling end-users to launch and configure completed distributed multimedia applications. Application descriptions and launching options are provided on the World Wide Web in the form of application pages to which Internet users can interact. These launching options allow the end-user to start a completely new application, import the modules of an already running application, or start a client portion and connect to an already running server portion.
As of this writing, all of the components of the application management system have been implemented. The liaison is implemented using Java and the other system components are implemented as Playground C++ modules. However, extended connection management capabilities of an upcoming version of the Playground run-time system is required to achieve full functionality.

A more detailed description of this application management system is provided elsewhere [44], [67].
Chapter 6

Usability Study

The usability of the EUPHORIA user interface management system (Chapter 2, Chapter 3) and the visual configuration language (Chapter 1) was evaluated through an empirical study of end-users. The majority of subjects were end-users with no prior experience with user interface construction or distributed application development. These end-users represent a wide variety of disciplines, including Political Science, Physics, Graphic Design, Electrical Engineering, Business, and Biological Engineering. Given representative tasks associated with distributed multimedia application construction, subjects were timed and evaluated to see how easily the various application construction techniques could be learned and applied. The series of tasks were designed with a target time of 75 minutes for the experimental session. For this reason, it was not possible to test every application construction technique described in this dissertation.

Primary Hypothesis: Through the use of the EUPHORIA user interface management system and the visual configuration language, first-time end-users of our software can learn and effectively apply the techniques of:

1. constraints,
2. constraint visualization,
3. imaginary objects,
4. configuration of distributed applications, and
5. distributed multimedia application construction.
Secondary Hypothesis: Novice end-users (i.e., end-users with little or no computer science training) can utilize the above application construction techniques as rapidly and accurately as expert end-users (e.g., computer science graduate students).

6.1 Design

The study consisted of five representative tasks associated with the construction of distributed multimedia applications. Each end-user was given a booklet of instructions (Appendix E) which introduced all of the necessary concepts and described the tasks to be performed. The instructions were organized as a series of five sections. Each section consisted of a number of practice exercises, designed to provide a hands-on introduction of the associated application construction techniques, followed by a task to be performed. The following subsections describe these practice exercises and tasks.

6.1.1 Constraints

The practice part of this section (Figure E-2) consisted of exercises introducing basic drawing techniques (e.g., drawing a rectangle) and end-user defined constraint relationships.

![Figure 6-1: Constraints task solution.](image)

In the task part of this section (Figure E-3), subjects were asked to create a drawing of "Mickey Mouse's" head. Subjects were instructed to make the ears and the head strictly circular, make the size of the ears equal, and attach the ears to the head. These requirements necessitated the use of constraints. Figure 6-1 shows the series of steps leading to one possible solution of this task (established constraints are shown as lines).
6.1.2 Constraint Visualization

The practice part of this section (Figure E-4) consisted of exercises introducing constraint visualization and deletion of constraints.

![Figure 6-2: Constrained drawing.](image)

In the task part of this section (Figure E-5), subjects were given a constrained drawing (Figure 6-2) and were asked to use constraint visualization to understand its behavior. The drawing consists of three rectangles. Rectangle 1 changes width and height whenever rectangle 2 or rectangle 3 is manipulated. Rectangle 2 and rectangle 3 change heights when rectangle 1 is resized. Subjects had to describe the constraint relationships among these rectangles. That is, the width and height of rectangle 1 is constrained to the heights of rectangle 2 and rectangle 3; the total height of rectangle 2 and rectangle 3 is constant because of anchor constraints on their corners.

6.1.3 Imaginary Objects

The practice part of this section (Figure E-6) consisted of exercises introducing imaginary objects.

![Figure 6-3: Imaginary objects task solution.](image)

In the task part of this section (Figure E-7), subjects were asked to create a "temperature controller" for controlling the proportion of "hot" (represented as a red rectangle) to "cold"
(represented as a blue rectangle). Subjects were instructed to make the heights of the blue and red rectangles inversely proportional (e.g., making the red rectangle taller automatically results in making the blue rectangle shorter), requiring the use of an imaginary object. Figure 6-3 shows the series of steps leading to one possible solution of this task (established constraints are shown as lines, anchors are shown as squares).

### 6.1.4 Module Configuration

The practice part of this section (Figure E-8) consisted of exercises introducing modules, published variables, and configuration.

![Figure 6-4: Module configuration solution.](image)

In the task part of this section (Figure E-9), subjects were given three running modules with descriptions, including a completed GUI constructed within EUPHORIA. To establish the behavior of the application and its GUI, subjects had to configure the communication among the modules using the visual configuration language.

The application was a planetary orbits simulation animating the path of the Earth and Moon around the Sun. An Earth module was supplied that generates the position and diameter of the Earth. A Moon module was supplied that computes the position of the Moon, relative to the Earth, and generates the position and diameter of the Moon. Subjects had to correctly configure logical connections among the Earth, Moon, and EUPHORIA display (Figure 6-4).
6.1.5 Distributed Multimedia Application Construction

The practice part of this section (Figure E-10) consisted of exercises introducing publishing variables in EUPHORIA and additional graphics object types.

![Diagram](image)

*Figure 6-5: Application construction task solution.*

In the task part of this section (Figure E-11), subjects were given three running modules with descriptions and were asked to create a distributed multimedia application. The application to be constructed was a multimedia message system: given the name of a person, entered graphically by application's user, the application displays their address, photograph, and current status. An address book module is supplied to retrieve addresses and photographs for individuals given their name. An answering machine module is supplied to retrieve a status and date for individuals given their name.

Subjects were asked to create a graphical user interface within EUPHORIA, to configure the communication among the three modules, and to interact with the interface. This task required an understanding of published attributes in EUPHORIA and their relationship to a distributed application’s configuration.

6.2 Subjects

Subjects were Washington University undergraduate and graduate student volunteers ranging from 18 to 30 years of age. The study was advertised by posting flyers around the Washington University campus. Subjects were compensated $10 for their participation.
A total of 31 subjects participated in the study. The prerequisite for participation was experience using commercial graphics editors (e.g., MacDraw), but no prior exposure to our software. Three subjects were eliminated from the statistical analysis because they had never used a graphics editor. The remaining 28 subjects represented a wide variety of disciplines: Biological Engineering, Business, Chemical Engineering, Computer Engineering, Computer Science, Electrical Engineering, English Literature, German, Graphic Design, Jewish & Near Eastern Studies, Math, Physics, Psychology, Political Science, and Systems Science.

6.3 Testing Procedure

Each subject performed the various tasks using our software on a Sparc workstation (a Sparc 5, Sparc 20, or an UltraSparc 1). The booklet of instructions (Appendix E) provided each subject with tutorials and descriptions of the tasks to be performed. Subjects had no previous exposure to our system and no additional instruction was provided.

An evaluation form was also given to each subject which included a brief questionnaire and space to write comments. Subjects were responsible for recording the times associated with each section (a clock was provided on the computer screen) and saving the task results (i.e., drawings and configurations) for later evaluation. The evaluation form also included a confidentiality agreement stating that they would not discuss the details of the study with future participants (each subject signed the agreement).

6.4 Scoring

Scoring of data was performed as follows. For each task, an accuracy was computed for each subject based on the percentage of the task completed, as summarized in Table 6-1.

In summarizing and comparing the subject times on a task, only the times of the subjects who completed the task with 100% accuracy were used. The reason for excluding the less accurate subjects is to get a more reliable measure of the task time, avoiding subjects who finished prematurely. All times for the practice and task portions include the time for the subject to read the associated instructions.
Table 6-1: Task Scoring.

<table>
<thead>
<tr>
<th>Task</th>
<th>Parts</th>
<th>Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>Drawing of Mickey Mouse.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Constraining the ears to the head.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Constraining the ears to be the same size.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Constraining the ears to be circular.</td>
<td>25</td>
</tr>
<tr>
<td>Constraint Visualization</td>
<td>Identifying anchor constraints.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Identifying constraint among rectangle 2 &amp; 3.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Identifying the width constraint.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Identifying the height constraint.</td>
<td>25</td>
</tr>
<tr>
<td>Imaginary Objects</td>
<td>Drawing the controller.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Anchoring rectangles properly.</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Using an imaginary object correctly.</td>
<td>50</td>
</tr>
<tr>
<td>Module Configuration</td>
<td>Correctly making each of 5 connections.</td>
<td>20</td>
</tr>
<tr>
<td>Application Construction</td>
<td>Drawing the user interface.</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Publishing the appropriate attributes.</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Module configuration.</td>
<td>35</td>
</tr>
</tbody>
</table>

6.5 Primary Hypothesis Results

For all sections, the majority of the subjects were able to complete the associated tasks within a reasonable amount of time. Table 6-2 shows the overall subject accuracy and times.

Table 6-2: Overall mean times and accuracy.

<table>
<thead>
<tr>
<th>Section</th>
<th>Practice Time (minutes)</th>
<th>Task Accuracy</th>
<th>% of Subjects Completely Accurate</th>
<th>Task Time for Subjects Scoring 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>4.9</td>
<td>86</td>
<td>64</td>
<td>7.9</td>
</tr>
<tr>
<td>Constraint Visualization</td>
<td>4.5</td>
<td>94</td>
<td>82</td>
<td>5.7</td>
</tr>
<tr>
<td>Imaginary Objects</td>
<td>3.1</td>
<td>84</td>
<td>75</td>
<td>11.3</td>
</tr>
<tr>
<td>Module Configuration</td>
<td>3.4</td>
<td>100</td>
<td>100</td>
<td>3.5</td>
</tr>
<tr>
<td>Application Construction</td>
<td>7.2</td>
<td>94</td>
<td>89</td>
<td>14.3</td>
</tr>
</tbody>
</table>

Table 6-2’s “Practice Time” column represents the mean practice time for all subjects. “Task Accuracy” is the mean accuracy of all subjects, including partial credit for incomplete tasks, as described in Section 6.4. The “% Completely Accurate” column represents the percentage of
subjects who completed each task with 100% accuracy. The mean task times for the 100% accurate subjects are listed in the “Task Time” column.

6.5.1 Constraints

The accuracy for the constraints task (Section 6.1.1) was a bit lower than we had anticipated. Ten subjects were unable to successfully complete this task. We conjecture that this is primarily due to the fact that it was the first part that involved “problem solving” rather than just following step-by-step instructions. Subjects only spent an average of 4.9 minutes learning the basic concepts and practicing before starting this task; further practice exercises probably would have resulted in a higher task accuracy rate. Also, it is clear that several subjects just did not fully comprehend the consequences of the instruction requirements (Figure E-3). These subjects claimed to have successfully completed the task, drawing an accurate looking picture. However, not all of the required constraint relationships were established in the drawing. This may indicate that subjects are accustomed to static drawings that do not change and are not used to dynamic drawings that can be manipulated.

6.5.2 Constraint Visualization

The time and accuracy ratings for the constraint visualization task (Section 6.1.2) were very good. Subjects found EUPHORIA’s constraint visualization mechanism to be very intuitive. Several people even complained that they would have done better in the first task if constraint visualization had been introduced earlier.

6.5.3 Imaginary Objects

The imaginary objects task (Section 6.1.3) was the most difficult task of the study, requiring the most problem solving skill. Even though more subjects successfully completed this task than the constraints task, the people who did not complete this task did not even attempt to use imaginary objects (a requirement to solve this task). Subjects only spent an average of 3.1 minutes practicing before starting this task; further practice exercises may have resulted in a higher task accuracy rate. Unlike the constraints task, the wording of the instructions was probably not a factor in the accuracy ratings.
6.5.4 Module Configuration

The module configuration task (Section 6.6.4) was completed with 100% accuracy by all subjects fairly rapidly. This seems to indicate that our visual configuration mechanism is especially effective.

6.5.5 Distributed Multimedia Application Construction

The application construction task (Section 6.6.5) was, perhaps, the most important part of this study. This section required the use of concepts from other sections, combining user interface construction techniques with distributed application configuration. The results are encouraging: subjects were able to construct and configure a complete distributed multimedia application in an average of 14.3 minutes. The mean accuracy score for this task was also excellent.

6.6 Secondary Hypothesis Results

For the secondary hypothesis' statistical analysis, the subjects were divided into three categories (Table 6-3) according to their amount of formal computer science training.

<table>
<thead>
<tr>
<th>Category</th>
<th>n</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
<td>8</td>
<td>People who have never taken a computer science course.</td>
</tr>
<tr>
<td>Intermediate</td>
<td>10</td>
<td>People who have taken 1 or 2 computer science courses.</td>
</tr>
<tr>
<td>Expert</td>
<td>10</td>
<td>People who have taken more than 2 computer science courses.</td>
</tr>
</tbody>
</table>

Table 6-4 lists the accuracy rates of the subject categories for the various tasks. For each task, the accuracy of the novice subjects is nearly as high as the accuracy of the expert subjects. The only notable exception is in the application construction task. All of the expert and intermediate subjects completed this task with 100% accuracy, but three novice subjects were unable to complete the task, resulting in a lower accuracy rating. Also note that there is not a strong trend among the differences between the three groups. Sometimes the novice subjects out-performed the intermediate subjects or the intermediate subjects out-performed the expert subjects.
Table 6-4: Category-based mean accuracy scores (percent correct).

<table>
<thead>
<tr>
<th>Task</th>
<th>Novice</th>
<th>Intermediate</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>81</td>
<td>88</td>
<td>83</td>
</tr>
<tr>
<td>Constraint Visualization</td>
<td>91</td>
<td>81</td>
<td>100</td>
</tr>
<tr>
<td>Imaginary Objects</td>
<td>75</td>
<td>90</td>
<td>88</td>
</tr>
<tr>
<td>Module Configuration</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Application Construction</td>
<td>79</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

The following subsections compare the times of the three categories on the various tasks using a one-way between-subjects analysis of variance test. As in Section 6.5, for each task, only the times of the 100% accurate subjects are used.

6.6.1 Constraints Task

The mean times for the three end-user categories in the constraints task were: novice, 10.6 minutes; intermediate, 8.7 minutes; expert, 6.8 minutes. Analysis of variance performed on the data indicated that there were no significant differences among the groups, $F(2, 15) = 1.18$, $MSe = 16.50$, $\beta=0.95$.

![Constraints task results](image)

Figure 6-6: Constraints task results.

Figure 6-6 shows the times of the three categories for the constraints task. In this chart, thin bars are used to show the time range and thick bars are used to show one standard deviation above and below the mean time. Nearly an equal number of subjects in each category were less than 100%
accurate (thus, were eliminated from the analysis and this chart). Although it appears that the times of novice category are much higher in comparison to the other categories, this is due to a single outlier value (20 minutes) that is nearly double of the next highest time. The standard deviation is also broad because of an unusually fast time in this category (3 minutes), which is actually the fastest time of all the subjects. We did not remove outlier values from our data since the sizes of the categories are relatively small.

6.6.2 Constraint Visualization Task

The mean times for the three end-user categories in the constraint visualization task were: novice, 5.7 minutes; intermediate, 5.6 minutes; expert, 5.8 minutes. Analysis of variance performed on the data indicated that there were no significant differences among the groups: $F(2, 20) = 0.02$, $MS_e = 6.43$, $\beta = 0.95$.

![Figure 6-7: Constraint Visualization task results.](image)

Figure 6-7 shows the times of the three categories for the constraint visualization task. Nearly an equal number of novice and intermediate subjects were less than 100% accurate (thus, were eliminated from the analysis and this chart); the expert subjects all completed this task with 100% accuracy. It is clear that there is almost no difference among the three groups on this task.
6.6.3 Imaginary Objects Task

The mean times for the three end-user categories in the imaginary objects task were: novice, 9.8 minutes; intermediate, 13.1 minutes; expert, 12.9 minutes. Analysis of variance performed on the data indicated that there were *no significant differences* among the groups: $F(2, 18) = 0.395, MS_e = 49.70, \beta = 0.95$.

![Figure 6-8: Imaginary Objects task results.](image)

Figure 6-8 shows the times of the three categories for the imaginary objects task. Nearly an equal number of subjects from each category were less than 100% accurate (thus, were eliminated from the analysis and this chart). Although it appears that the expert subjects’ times are actually much higher than the other categories, this is due to a single outlier value (31 minutes) of a very persistent computer science graduate student. Also, this category contains the fastest two times (a tie, 5 minutes) resulting in a fairly broad standard deviation.

6.6.4 Module Configuration Task

The mean times for the three end-user categories in the module configuration task were: novice, 6.3 minutes; intermediate, 3.5 minutes; expert, 2.8 minutes. Analysis of variance performed on the data indicated that there were *significant differences* among the groups: $F(2, 25) = 7.21, p < .01, MS_e = 3.98$. 
Figure 6-9: Module Configuration task results.

Figure 6-8 shows the times of the three end-user categories in the module configuration task. All subjects were able to successfully complete this task. Clearly, the times for the novice subjects are greater than the times of the intermediate and expert categories. The times of the intermediate and expert categories, however, are not significantly different. The intermediate and expert categories consisted largely of computer science and electrical engineering majors; the novice category consisted mainly of non-technical majors. The difference is probably because people in computer science and electrical engineering routinely deal with component-based computation abstractions and people in non-technical majors do not have this exposure. But it should also be noted that one person in the novice category solved this task in only 2 minutes and was in a fairly non-technical area of study (German).

We believe that with additional visual configuration experience, novice end-users could configure distributed applications as quickly as expert end-users. After all, the novice end-users only spent an average of 3.5 minutes practicing for the module configuration task.

6.6.5 Distributed Multimedia Application Construction Task

The mean times for the three end-user categories in the distributed multimedia application construction task were: novice, 14.6 minutes; intermediate, 19.1 minutes; expert, 12.3 minutes. Analysis of variance performed on these data indicated that there were no significant differences among the groups: $F(2, 22) = 3.37$, $MS_e = 35.01$, $\beta = 0.3$. 
Figure 6-10: Application Construction task results.

Three subjects in the novice category were less than 100% accurate (thus, were eliminated from the analysis and this chart). While one of these subjects only made a minor mistake, the other two did not successfully construct the application. We believe that phrasing of the instructions (Figure E-11) may have intimidated these individuals (e.g., “Create a user interface”). This, combined with fatigue, resulted in these subjects quitting prematurely.

Figure 6-8 shows the times of the three end-user categories in the distributed multimedia application task. It is clear that there is little difference between times of the novice, intermediate, and expert subjects.

6.7 User Feedback

In general, the subjects of the study enjoyed using EUPHORIA and the visual configuration language. The main complaint was that EUPHORIA does not have an “Undo” mechanism. The following are some end-user quotes taken from the evaluation form comments:

“I hadn’t seen constraint-oriented graphics before and it was really easy to grasp.”

“...when a handle is being pointed to, it should become shadowed or something. I often missed when I tried to select a handle, and since I didn’t know if I had succeeded or not, I had to backtrack, checking my constraints until I found the one I missed.”
"The most difficult part was getting the look right. The actual configuration part was very easy. Imaginary objects are very helpful."

"My brain hurt for the first minute thinking about it [temperature controller], but once I anchored the black bars it was easy... I started with only one imaginary object and then putzed around and saw the problem. It was never frustrating, though."

"Cool! Really neat graphics."

"Constraint-based interface is easy to use but it's hard to imagine beforehand what a particular constraint will do."

"Interesting. This is a cool concept" [imaginary objects]

"The 'show constraints' idea seems very helpful, but it could become confusing with complicated drawings."

"It's a good system. I especially like the interaction between EUPHORIA and the Connection Manager."

"I like EUPHORIA. It has a lot of neat features I've never seen before."

"I was having far too much fun playing with this one!" [application construction]

6.8 Summary

To test the effectiveness of both the EUPHORIA software and our visual configuration language, we conducted a usability study in which end-users performed tasks representative of constructing distributed multimedia applications. A total of 28 undergraduate and graduate Washington University students from a variety of disciplines participated in this study. The results showed that end-users were able to construct and configure complete distributed multimedia applications. Further, for all of this study's user interface construction tasks, end-users with no formal computer science background were able to perform as well as experienced computer programmers.
Chapter 7

Conclusion

With today’s emerging interest in the Internet among the general public, it is vital to provide intuitive graphical interfaces for distributed multimedia applications. No matter how fast networking technology becomes, mainstream end-users will not utilize this technology unless it is easy to use. After all, the Internet has been in existence for over 20 years, providing services such as ftp and gopher. It only became popular commercially and publicly when the World Wide Web and graphical browsers were developed, resulting in an unprecedented growth in Internet content and use over the past couple of years. This dissertation’s goal was to facilitate the development of distributed multimedia applications, enabling both end-users and researchers to create customized applications for their individual needs.

We made progress toward this goal by providing a number of application construction techniques from the areas of graphical user interface construction, constraints, and application management. With these application construction techniques, end-users can assemble off-the-shelf application components, create direct manipulation graphical user interfaces, and use completed applications. This approach allows distributed multimedia applications to be developed more rapidly, many times by the people who will actually use them, reducing the dependence on computer programmers. Giving these people this creative power will ultimately result in more diverse applications that are better suited to particular problem areas, satisfying the needs of end-users more effectively.

The following sections discuss the class of applications that can currently be constructed with our application construction techniques, some experience with constraints, and future research directions.
7.1 User Interfaces that Can and Cannot be Constructed

Many previous software systems have been developed to aid the construction of "dialog box" style user interfaces. The NeXT interface Builder [53], UIMX [79], and Prototyper [68] are some examples. These systems provide a "point and click" approach using a number of standard graphical mechanisms (e.g., buttons, menus). In contrast, this dissertation focused on the creation of application-specific graphics rather than these standard graphical mechanisms. For this reason, most dialog box style user interfaces cannot currently be constructed with our software.

Our work focused on direct manipulation of state information. For example, dragging a graphics object manipulates its position. Event-based interaction such as clicking in a particular location within a graphics shape or key commands (i.e., other than changing a text field) are not supported. However, in the future additional graphics object attributes (e.g., a "selected" attribute) will be added which will support some types of event-based interaction.

The following is a list of some kinds of applications for which user interfaces can be constructed using our construction techniques:

1. Multimedia Applications
2. Multi-user Collaborative Work
3. Process Control
4. Algorithm Visualization
5. Simulation
6. Parallel Processing Applications
7. Client/Server Applications

7.2 Experience with Constraints

Constraints are powerful since they allow relationships to be defined declaratively rather than programmatically. This allows new relationships to be added to an existing system fairly easily since the constraint solver automatically handles the dependencies among the constraint variables.
However, complex constraint relationships can be very difficult to debug. Specifying relationships programmatically has an advantage when it comes to debugging, since existing debuggers can be used to single-step and examine the consequences of state changes within the (static) source code. In contrast, constraints are dynamically changing graph-based data structures. Conventional debugging techniques are not very helpful since the overall computation topology is not in a form that can easily be examined. Although EUPHORIA’s constraint visualization (Section 2.5.2) and other work in showing constraints graphically [61] helps, debugging large constraint systems is still very difficult.

![Diagram of constraint behavior](image)

**Figure 7-1:** Constraint nondeterministic behavior.

From the end-user’s perspective, constraints can be confusing at times. Constraint systems can be *under-constrained*, resulting in nondeterministic behavior since different choices can be made arbitrarily by the constraint solver. For example, we have a simple demonstration that increments the value of an integer published variable; connecting this variable to the width of a rectangle results in the rectangle growing over time (Figure 7-1). The choice of whether to grow the rectangle to the left, to the right, or in both directions simultaneously from the center is arbitrary. Suppose that the system decides to grow to the right. The end-user can override this decision by anchoring the top-right and bottom-right points of the rectangle, resulting in the rectangle growing left. One might expect that removing these anchors would result in the rectangle resuming its growth to the right. However, this is not the case. Since the growth direction is arbitrary and the anchors had previously forced the rectangle’s underlying constraints to choose growing left, the system has no reason to override this decision. Therefore, the rectangle continues to grow to the left even after the anchors have been removed.

A partial solution to this problem would be to add “stay” constraints of various strengths to each the rectangle’s attribute variables. This would result in more predictable behavior since each attribute would have an importance associated with it relative to the other attributes. For example,
in choosing which direction to grow the above rectangle, the least important attributes (or most important, depending on how strengths are specified) is chosen to be computed by the constraint system. However, nondeterminism cannot be totally avoided since the end-user can still specify underconstrained constraint relationships. We did not include these stay constraints in EUPHORIA because, in general, it is not clear that certain attributes are always more important than others. Furthermore, end-users can achieve the same effect through the use of anchors and other constraints of various strengths, customizing the behavior for particular applications.

Although constraints are actively being studied by many people, we are not aware of any commercial systems that utilize constraints. A combination of the above problems as well as reliability issues (e.g., not always being able to enforce all constraints in a system) are most likely the reason. Section 7.3.2 describes an approach that would make constraint solving more reliable.

7.3 Research Directions

This section describes some possible research directions that build upon this dissertation.

7.3.1 Visual Programming

Calculators encapsulated inside widgets would allow frequently used equations to be easily duplicated. Figure 7-2 shows an addition widget, built using a calculator object. By constraining handles to the three variables, one could constrain a value to be the sum of two other values.

![Figure 7-2: An addition widget.](image)

Apart from the definition of user interface modules, we have not addressed the problem of end-user construction of the computational components. A visual programming language supporting the construction of simple Playground modules would offer the user increased flexibility in the
construction of custom applications. A promising approach to this problem would be the integration of Playground with a general purpose visual computation language based on dataflow concepts, such as the “Show and Tell” system [33].

7.3.2 Hybrid Constraint Algorithm

Constraint algorithms that maintain cycles of constraints can encounter unsolvable series of equations (Section 4.2.3), leading to unpredictable results. UltraBlue’s cycle avoidance technique (Section 4.4.3) provides a fast, yet reliable way to solve a series of hierarchical constraints that can potentially have cycles of constraints. However, UltraBlue may leave some constraints unenforced in order to avoid cycles. In some cases, these constraints could have only been solved by using a simultaneous equation solver (i.e., as is generally utilized by algorithms that maintain constraint cycles).

Equations:

\[
\begin{align*}
B &= A - 5 \\
B &= C \\
D &= C - 10 \\
A &= C + D
\end{align*}
\]

![Figure 7-3: Constraints requiring a simultaneous equation solver.](image)

For example, Figure 7-3 shows a series of constraints that cannot be satisfied with a cycle avoidance strategy. In this example, each constraint is of equal importance so it is desirable to enforce all the specified constraints. However, in enforcing all of the constraints, two cycles arise among the \(A, B, C, \) and \(D\) constraint variables. Given an initial value for these variables, evaluating the constraints on the cycles iteratively results in inconsistent variable values. If \(A\)'s initial value is 0 then \(B, C, \) and \(D\) are computed to be -5, -5, and -15 respectively. Since \(A = 0\) and \(A = C + D,\) there is an inconsistency. However, a simultaneous equation solver would produce a correct solution of \(A = 20, B = C = 15,\) and \(D = 5.\)
A solution to the problems of cycle maintaining and cycle avoidance algorithms would be to develop a "hybrid" constraint algorithm that combines the approaches of both cycle maintaining and cycle avoiding algorithms. The hybrid algorithm would be capable of analyzing a series of interdependent constraints associated with a cycle and determining whether to maintain or avoid the cycle. Since maintaining and solving cycles of hierarchical constraints is NP-complete [39], cycle avoidance would be the preferred approach. However, if it is determined that more constraints can be enforced when cycles are maintained, and the constraints’ equations are solvable in this way, then the cycle maintenance approach would be applied.

### 7.3.3 Aggregate Mappings Layout Rules

It would be helpful to have an end-user mechanism to specify the layout of aggregate mapping instances. This mechanism would enable a further decoupling of applications from their GUIs by eliminating the need for the implementor to specify element position information in the application’s code. One approach to this problem is to specify the layout graphically through induction-style rules. With this strategy, similar to a demonstrational approach [13], [14], [46], the end-user would specify the relationship between the first aggregate element (base case), an aggregate instance \( k \), and the next element \( k+1 \). From this information, the system would inductively place each instance element of the aggregate. In addition, a number of boundary conditions could be included to deal with special cases.

For example, Vaudeville’s user interface (Section 5.2.2) was constructed during the early stages of EUPHORIA's development, and therefore did not utilize EUPHORIA. However, one could imagine constructing this GUI with EUPHORIA, using aggregate mappings to display the participant member’s pictures. Figure 5-2 displays six participants in a table of dimensions three by two. Currently, it is necessary to explicitly include the position of each of these participants in the application portion, informing EUPHORIA how to arrange the aggregate mapping instances. This is inconvenient since it ties the application code to the specifics of the GUI's visual appearance. Using induction-style rules, one could specify the layout among the elements taking into account the size of the containing space (for wrapping). In this way, each element of the aggregate is displayed in a standard table configuration based on the size of its space.
7.3.4 Integrated UIMS and Visual Configuration Language

The module visual abstraction described in Section 1.5.1 is an intuitive mechanism for configuring Playground applications. However, for large applications it can become cumbersome to deal with many interconnected modules in this way. We have a number of enhancements to the module visual abstraction planned.

Currently, modules are configured in a separate user interface called the Connection Manager Front-End, which was developed nearly a year before the start of EUPHORIA's development. In the future, we will integrate the visual configuration functionality into EUPHORIA. This will allow us to leverage off the existing mechanisms to support a more sophisticated configuration tool. Collections of modules could be partitioned into different groupings through the use of alternatives. Encapsulated modules (i.e., modules containing other modules) could be created through the use of widgets. Also, with widgets the appearance of each module could be customized by the end-user. For example, a module that sends video from a camera could be represented using a widget with a camera icon. Finally, aggregate mappings could be used to automatically visualize the set of modules and their connections, eliminating the need for special-purpose coding of this interactive visualization.

Encapsulated modules would make our system scale better, greatly simplifying the visual configuration display. For example, the filter modules described in Section 5.2.1 could have been encapsulated into a single filter module [19]. A single module would make Figure 5-1 easier to comprehend. One would be able to place the “Dispatch,” “Spatial,” “Temporal,” and “Threshold” modules into a widget representing the overall filtering operation, publish the external variables of the widget, and make the internal pipeline modules imaginary (i.e., hide the details of the computation).

7.4 The Future

The future of computing is likely to be a time of great expansion, attracting many novice end-users who had previously never imagined that they would want to use computers. The early stages of this trend can already be seen in the popularity of the World Wide Web, providing information and
entertainment services not only to technically sophisticated "nerds" but also to a broad range of people in the general public. The reason for this success is obvious: web browsers provide intuitive graphical user interfaces that people like to use. We believe that this will prove to be a stepping stone on the way to initiating the general public into the world of computing. According to CNN, last year was the first time that more personal computers were sold than televisions, the primary source of entertainment and news for the general public. As interest grows, people will demand more interactive, customized applications. Many individuals will want to at least partially construct these applications rather than relying on programmers. Our research in user interface construction and visual configuration provides a means to meet this challenge.

In addition, we believe that the nature of software distribution will also change significantly in the future. Currently, if one wants to use an application such as a word processor or spreadsheet at home, one must go to the store, buy the disks or CDs containing the software, and install the software on their local hard drive. When the software becomes outdated later, the individual is often forced to buy and install an upgrade in order to stay compatible with their (ever changing) operating system and window system. This requirement is both inconvenient and expensive. With the emergence of high speed networks to the home, it is unlikely that end-users will be forced to endure this process in the future. Instead, it is likely that people will use applications remotely on a "pay-per-use" basis, either subscribing to an application or set of applications on a monthly basis or paying time based rates (e.g., 5 cents per minute). In this way, people will always have the state-of-the-art versions of software available to them at a moment's notice without a major software investment. This approach could also take advantage of distributed computing technology. That is, one could use multiple (remote) computers concurrently to accomplish tasks faster. It is hoped that the contributions of this thesis will one day help to make this vision of the future a reality.
Reconfiguring a Timer Server Clerk

• Terminating a clerk

    remove Time_Server_Clerk

• Reconfigure a new Clerk

    dynamic Passive_Time_Server_Clerk Service *
    libnet_svcs_p.dll:make_Clerk()
    "-h tango.cs:$TIME_SERVER_PORT"
    "-h perdita.wuerl:$TIME_SERVER_PORT"
    "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
the probability that \( x_t \geq 1 \) is at most \( \epsilon \). For \( n = 1000, \epsilon = 10^{-5} \) and \( d = 6 \), this implies that with probability \( 1 - \epsilon \), all endpoints receive the packet after two transmissions. For \( n = 10^6, \epsilon = 10^{-4} \) and \( d = 10 \), four transmissions are enough. For \( n = 10^9, \epsilon = 10^{-3} \) and \( d = 25 \) eight transmissions are enough. Thus, while the proposed mechanisms fail to achieve scalability in the strictest sense, for most practical purposes they are as good a truly scalable algorithm.

7 Closing Remarks

In this report, we have not addressed issues of flow control and congestion control, in the reliable multicast context. For a one-to-many reliable multicast, one can use adaptive windowing techniques like those used in point-to-point protocols to adapt the sender's rate to accommodate slow receivers and/or congested links. Similar techniques are applicable to a many-to-one connection. The use of flow/congestion control on a one-to-many multicast does have the effect of slowing the connection data rate to that of the slowest receiver or most congested link. For distributed computing applications, this may well be the right thing to do. For information distribution applications, it's not clear that this is an appropriate approach. Indeed, there may be no single approach that is really suitable for all applications, as argued in [7].

If one is not concerned with the provision of consistent delivery order to all receivers in a reliable multicast connection, it is possible to directly combine a many-to-one connection with a one-to-many connection, eliminating the relay node. This requires extending the one-to-many mechanism to recognize different subchannels. The main cost of this extension is that for each of the 15 subchannels, the VXT entry must keep track of which transmission slot the last start cell on that subchannel specified (so that the data cells for the subchannel can be forwarded to the proper downstream branches and so acknowledgements for the subchannel can be handled correctly). With a two byte transmission slot number, this overhead is 30 bytes. Because this allows multiple sources to send packets directly into the one-to-many connection, sources that are sending information concurrently must use distinct transmission slot numbers. Slot number ranges can be assigned statically when the number of sources is small, or they could be dynamically. The main drawback with this approach (in addition to not ensuring consistent ordering) is that it requires every receiver to maintain protocol state for all the active senders, rather than just the relay node.

References


Appendix A

Color Figures

This appendix contains color versions of selected figures, which are also contained within the text. Each figure repeated here is denoted by an asterisk by the figure title within the text. If possible, this appendix should be printed on a high resolution color printer.
Figure 1-1: Orbital simulation modules.

Figure 1-4: Interactive orbital simulation in EUPHORIA.
Figure 1-5: Hypervideo Browser in EUPHORIA.

Figure 2-1: Maple syrup factory GUI in EUPHORIA.
Figure 2-2: Remote Control Video Camera GUI in EUPHORIA.

Figure 2-6: End-user creation of an adjustable picture frame using constraints.
Figure 3-1: Image morphing application in EUPHORIA and its output.

Figure 3-2: Distributed minimum spanning tree in EUPHORIA.
Figure 5-1: Medical image processing pipeline.

Figure 5-2: Vaudeville Video Teleconferencing GUI.
Figure 5-4: Application page for Vaudeville.

Figure 6-5: Application construction task solution.
Appendix B

EUPHORIA Reference Manual

The Programmers' Playground is a software library and runtime system for creating distributed multimedia applications [19], [20]. EUPHORIA is the user interface management system for Playground, enabling end-users to create direct manipulation graphical user interfaces (GUIs) for distributed applications [42]. EUPHORIA has an intuitive graphics editor that allows end-users to simply draw GUIs (see Figure B-1). The behavior of GUIs is established by forming logical connections between EUPHORIA and external Playground modules [20]. This document summarizes the features of EUPHORIA and how it can be used to create GUIs. It is meant as a companion to the Playground reference manual [20].

Figure B-1: EUPHORIA graphics editor, displaying an interactive maple syrup factory GUI.
EUPHORIA is meant to be controlled by a three button mouse. Note that throughout this document, whenever the words “click” or “drag” are used, the left mouse button is implied.

### B.1 Retractable Panes

The EUPHORIA window is divided into a number of panes (see Figure B-1), the tool palette, data boundary, alternatives interface, and main drawing area. The size of the panes can be adjusted by dragging the separator line of the pane. This is useful for hiding the tool palette and data boundary when a GUI is completed.

### B.2 Drawing

Drawing in EUPHORIA is much like drawing in other graphics editors such as MacDraw or FrameMaker. The EUPHORIA window contains a graphics tool palette and a main drawing area. With the tool palette, users can select shapes to be drawn, change the color of shapes, and perform other operations through the menus (see Figure B-2).

![Figure B-2: Tool palette.](image)

Once a drawing tool is selected, the shape corresponding to that tool may be drawn in the main drawing area. When drawing is complete, the drawing tool becomes unselected. Double clicking on a tool prevents the tool from becoming unselected after the first drawing. In this way, users can conveniently draw multiple shapes. When finished, clicking on a selected tool unselects it.

The color of a shape can be selected by clicking on the appropriate color entry below the shape palette. Clicking on a color entry changes the color of the selected graphics objects in the main drawing window and sets the color for all future drawings. Double clicking on a color entry changes the background color of the EUPHORIA editor.
B.2.1 Selection & Handles

In the main drawing area, clicking on a graphics object causes it to become selected or unselected. Multiple graphics objects can be selected at the same time; selecting a graphics object does not unselect other graphics objects. When no drawing tools are selected, dragging a selection box in the main drawing area will select all graphics objects within the box. Note that all previously selected graphics objects become unselected when dragging a selection box. Selected graphics objects can be deleted by pressing the backspace or delete key.

![Selected graphics objects and their handles.](image)

Figure B-3 shows some of the basic shape types and their selection handles. As with other graphics editors, most of these handles can be dragged to change the attributes of their graphics object. The color of each handle represents the type of information that it represents. For example, real number values such as width and height appear in blue; “point” x,y coordinate values appear in green. These handles are used not only for direct manipulation, but also for forming constraints among graphics objects, as described in Section B.4. Some handles are exclusively used for forming constraints. For example, the handle in the bottom middle of a text object is used to connect to its string attribute (see Section B.4.3).

B.2.2 GIF Images

GIF images may be loaded into the main drawing area by choosing Load GIF Image... from the “File” menu. A GIF image is treated much like a rectangle. Users change move, resize, and form constraints with GIF images. An image loaded in this way is actually a “movie” having a single frame as described in Section B.5.1.
B.2.3 Layering

Graphics objects have an associated layer attribute which controls the order in which graphics objects are drawn (i.e. which shapes are in front of other shapes). When a graphics object is created, it is set to the front-most layer. The layer can be changed with the “Layer” menu, allowing one to Bring to Front or Send to Back selected graphics objects.

B.2.4 Coordinate System

The coordinate system of the main drawing area is oriented with the x-coordinate axis increasing to the right, and the y-coordinate axis increasing downward (see Figure B-4). By default, the origin is at the top left corner of the main drawing area.

![Coordinate system & origin controller](image)

Figure B-4: Coordinate system & origin controller.

The position of the origin (i.e. the (0, 0) coordinate) and the scaling factor of the x and y axes may be modified through the origin controller (see Figure B-4). The origin controller is invoked by choosing Origin Controller from the “Edit” menu. Dragging the mouse within the drawing area sets the position of the origin to the mouse position. Clicking on an axis allows one to enter a new coordinate value. The entered coordinate value is used to determine a new scaling factor for the axis. For example, setting a value on the x axis to a high number increases the scaling factor, making all graphics objects shorter.

The origin controller provides a means to set the coordinate system of a drawing to convenient local coordinate units of an external application rather than raw pixel values.
B.3 Data Boundary

Perhaps the most apparent difference between EUPHORIA and other graphics editors is the ability to connect attributes of graphics objects to external modules. Changes to these attributes are sent to and from external Playground modules and EUPHORIA. These connections can be used to create animated visualizations and interactive, direct manipulation GUIs.

As described in [21], each Playground module (including EUPHORIA) has a set of “published” externally readable/writable variables called the *data boundary*. The data boundary is essentially an interface to the outside world.

![Diagram of data boundary]

Figure B-5: Data boundary.

Figure B-5 shows the graphical representation of the data boundary that appears as the left portion of the EUPHORIA window. The top portion of the data boundary contains an “external update control” button. This allows users to enable or disable communication between EUPHORIA and external modules. It is sometimes useful to turn off communication for a period of time so that animation can be suspended, allowing graphics objects to be modified. Although it is possible to edit animated graphics objects with communication activated, it is sometimes hard to grab a quickly moving object!

---

1. In other publications, the term “presentation” has been used. The term “data boundary” is used to avoid confusion with graphical presentations.
B.3.1 Published Variables

A graphics object handle can be published, meaning that the attribute that it controls is connected to an externally readable or writable published variable. This is achieved by dragging, with the *middle mouse button*, a connection line from a graphics object handle to the "new variable area" of the data boundary. This has the effect of creating a visual representation of a published variable, informing Playground’s connection manager of the variable, and forming a constraint (see Section B.4) between the handle's graphics object attribute and the published variable. Similarly, constraints can also be formed between graphics object attributes and variables already in the data boundary, as described in Section B.4.2.

A published variable represents a value that is shared with external Playground modules. When a variable is changed in an external module, Playground sends the change out to all connected modules, including EUPHORIA. Similarly, when a graphics object is changed (e.g. moved by the user), this change may also be sent out to external Playground modules, according to the published variables and the logical connections between variables [20]. Figure B-6 shows the visual appearance of different types of published variables. As with handles, the color of a published variable represents its data type.

![Figure B-6: Published variables.](image)

Clicking on a published variable selects/unselects it. Pressing the backspace or delete key removes selected variables from the data boundary\(^2\). Double clicking on the name of a variable allows the name to be edited.

---

2. Be careful, selected graphics objects are also deleted. It is not possible to directly reinsert a published variable into its former location because of how the connection manager operates.
B.3.2 Variable Attributes Window

Double clicking on a variable opens a dialog box for viewing and changing its properties (see Figure B-7). Users can change the name, strength, protections, and other attributes of a variable.

![Variable Attributes](image)

Figure B-7: Variable attributes window.

A variable's strength affects how the system interprets updates from external applications. The strength represents the relative importance of its external updates in relation to connected constraints and user interactions. For example, by default, user actions such as dragging a graphics shape take precedence over updates from a published variable (i.e., a variable that is connected to the graphics shape). If the strength of the variable is set sufficiently high, updates from the variable take precedence over user actions. In the case when the variable is only write world, this means that the user cannot move the graphics shape; it can only be moved by external modules.

Each variable has protections that control the read/write permissions of the variable to external modules [20]. Clicking on an arrow toggles its protection on or off. Note that having only write world protection is treated as a special case which allows external updates to the variable to be processed in the internal constraint network more efficiently.

If the variable's "Skip Values" attribute is enabled (default), some intermediate values of the variable may be disregarded. This happens when external modules transmit values to EUPHORIA
faster than it can process the values, causing the Playground run-time system to queue multiple values for a single variable. The Skip Values attribute skips all old values in the queue and only uses the most recent value.

B.3.3 Add Variables Window

Tuple data types can be created which consist of multiple fields of data. Also, other data types such “character” and “boolean” can be published, allowing users to visualize and edit any Playground base type, tuple of base types, tuple of tuples, etc.

![Add Variables Window](image)

Figure B-8: Add variables window.

Double clicking on the “new variable area” of the data boundary shows the “Add Variables Window” (see Figure B-8). Any data type listed in this window can be published by selecting the type, entering a name, and pressing the Add button.

Creating Tuple Types

The set of variables in the data boundary can be “captured” as a group. Pressing the Capture button creates a new tuple type with the data boundary variables as the fields of the tuple and the entered name of the Add Variables Window as the type name. This new type is inserted into the list of available types.
Variables of a user defined type can be published by pressing the "Add" button just as with other types. When published into the data boundary, a tuple appears in green with a small triangle to the left. This triangle is used to expose or hide the fields of the tuple (see Figure B-6).

Creating Array Types

Array of tuple data types can be constructed. Pressing the Array... button in the Add Variables window brings up a window for creating array types. Within this window, one can specify the tuple element type, array dimensions (up to four are allowed), and size of each dimension. Types created this way are added to the Add Variables window list of types.

B.4 Constraints

Users can establish constraint relationships among the graphics object attributes, published variables, and individual tuple fields. In addition to direct manipulation, the graphics object handles are also used in forming constraint relationships. A constraint is a persistent relationship between graphics object attributes. Once a constraint is formed, the system is responsible for maintaining the relationship when changes are made to graphics objects or published variables. Four types of constraints are currently supported: anchor, equality, conversion, and formula.

B.4.1 Anchor Constraints

Clicking on a handle with the right mouse button causes the corresponding attribute of the graphics object to become constant. So, for example, if a rectangle's width and height handles are set to be constant, the size of the rectangle becomes fixed. Clicking with the right mouse button on the handle a second time releases the anchor constraint.

B.4.2 Equality Constraints

An equality constraint can be established by dragging a connection line between two handles of the same type with the middle mouse button. For example, a rectangle can be constrained to be a square by forming a constraint between its width and height handles. Equality constraints can also be formed between graphics object handles and published variables. Like equality constraints between handles, this is achieved by dragging a connection line, with the middle mouse button,
between a handle and a published variable. Note that publishing variables as described in Section B.3 automatically forms a constraint relationship.

Constraints to published variables are a means for visualizing and interacting with the value of a published variable. For example, one can form an equality constraint between the top-left handle of a rectangle and a point type published variable. Whenever the point variable is changed externally (i.e. from a separate module that is connected to the variable) the change is communicated to the rectangle, moving the rectangle to the appropriate position in the window. Similarly, whenever the rectangle is moved through direct manipulation, its updated position is sent out to the connected, external Playground modules.

### B.4.3 Conversion Constraints

Equality constraints can be made between handles or published variables of different types. These types of constraints are known as conversion constraints, since some kind of type conversion is necessary. For example, a real handle such as the width of a rectangle can be connected to an integer published variable. This results in a rounding operation when the real value is communicated to the integer published variable. Table B-1 lists the supported connection types. Note that only a subset of these conversion operations are available in Playground's connection manager for making connections among modules.

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th>integer</th>
<th>boolean</th>
<th>string</th>
<th>memory Block</th>
<th>tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>E</td>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>C</td>
<td>E</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td></td>
<td>E</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory Block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>tuple</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E/C</td>
</tr>
</tbody>
</table>
The conversion operation from string to boolean translates the following strings as having a boolean value of false: 0, f, F, false, False, FALSE. Every other string is interpreted as having a boolean value of true.

Tuples are compatible based on the number and types of the tuple fields (recursively). For example, a tuple with two real fields is compatible to a tuple with two integer fields. A tuple with two real fields is not compatible with a tuple with three real fields.

B.4.4 Constraint Strengths

In general, it is not always possible to satisfy every constraint in a series of constraints. Conflicting or cyclic constraint relationships may be specified, forcing the constraint solver to leave some constraints unsatisfied. To help the constraint solver how to decide which constraints will be satisfied and unsatisfied, each constraint is assigned a preference level called a strength. Strengths can vary from weakest to strongest, and can be set by the user in order to customize behavior within EUPHORIA.

![Default Strengths window](image)

Figure B-9: Strengths window

Figure B-9 shows the "Default Strengths" window, activated by choosing Set Default Strengths... in the "Constraint" menu. In addition to user defined equality and anchor constraints, the system also uses constraints internally in propagating incoming Playground variable values and manipulating graphics objects.
For example, by default, anchor constraints are more important than variables and dragging constraints. With these settings, if the position of a shape is anchored then the position cannot be changed by user dragging or by external updates from Playground variables. However, one could change the strength of a published variable connected to the shape's position to make its strength stronger than the anchor. The result would be that changes from external modules would change the position of the shape but user dragging would still not affect the position.

B.4.5 Constraint Visualization & Editing

Constraints can be visualized and edited. In the “Constraint” menu, choosing Show Constraints enables constraint visualization. Equality constraints are shown as flashing arrows between the handles of selected objects and/or published variables if the handles are sufficiently far apart. When handles overlap, equality constraints are shown as circles. Anchors are shown as squares. Unsatisfied constraints are shown as dashed shapes. Note that a visualization arrow can represent multiple constraints, in the case of tuples. For example, forming a constraint between two points actually forms two constraints: one between the x coordinates and one between the y coordinates. Double headed arrows are used to show the mixed computation directions.

By default, certain constraints are not shown. This includes constraints to imaginary objects (see Section B.5.2), and constraints in which at least one endpoint is not visible within the window. Choosing Show Hidden Constraints will show all constraints.

Sometimes different handles and constraints may be packed close together and thus difficult to visualize. By choosing Taffy Pull Mode from the “Constraints” menu, users can view constraints by stretching apart constrained graphics objects. In this mode, graphics objects move as if there were no constraints imposed, allowing users to freely manipulate them. However, the constraints are still visualized as described above. When the mode is disabled, all graphics objects snap back into place according to the established constraint relationships. As a short-cut to choosing the menu item, users can pull apart graphics object in the same way while holding down the Shift key.
A visualized constraint may be deleted by clicking on it with the right mouse button. Clicking on a constraint with the middle mouse button reveals a pop-up menu that can be used to change the strength of the constraint.

B.4.6 Formula Constraints

With a calculator object one can specify formula constraint relationships among graphics objects. Like other graphics objects, a calculator has a number of attributes, which can be used to make constraints. The attributes are given names, and an editing area allows the one to specify an algebraic formula using the attribute variables. A multi-way constraint graph is constructed from the formula, providing a means to compute any of the variables dynamically in terms of the others. After construction is completed, the calculator can be made imaginary (i.e., hidden, see Section B.5.2).

For example, a calculator object could be used to convert between scales of measurement, such as Celsius and Fahrenheit temperatures (see Figure B-10). The calculator maintains the mathematical relationship between these two variables, computing degrees Celsius when the thermometer is manipulated or computing degrees Fahrenheit when a new Celsius value is entered.

B.5 Advanced Drawing

EUPHORIA supports a number of high level mechanisms for constructing GUIs, including imaginary objects, alternatives, widgets, and aggregate mappings.
B.5.1 Movies

The movie tool in the toolbar (Figure B-11) represents the creation of a movie graphics object. A movie consists of a series of frames (i.e., images), with one frame displayed at any given time. When drawn, a movie is initially empty and is shown as a gray box. The movie's data handle can be published as an image tuple, allowing external modules to store frames in the movie. An “ID” field of an image is used to determine whether to replace existing movie frames (i.e., if an image arrives at the movie with ID=x, it will replace frame #x). The movie's frame # handle is used to specify which frame, according to the ID, to display.

![Diagram of Movie Tool and Movie](image)

Figure B-11: Movies

Note that it is usually necessary to disable the “Skip Values” attribute of the image published variable associated with a movie. Otherwise, some image frames may be skipped and will not arrive at the movie.

B.5.2 Imaginary Objects

Any graphics object can be made imaginary. An imaginary graphics object is not normally visible or selectable by the user. However, the underlying constraint relationships of an imaginary graphics object are maintained. In this way, imaginary objects are a convenient means for forming indirect constraint relationships among graphics objects.

A selected graphics object can be made imaginary by choosing Set or Unset Imaginary from the “Layer” menu. Imaginary graphics objects can be shown (which is useful for editing) or hidden by selecting Imaginaries Shown from the “Layer” menu.