Report Number: WUCS-95-19

1995-01-01

EUPHORIA Reference Manual

Authors: T. Paul McCartney and Kenneth J. Goldman

EUPHORIA Reference Manual

T. Paul McCartney and Kenneth J. Goldman

WUCS-95-19

July 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

EUPHORIA Reference Manual

T. Paul McCartney and Kenneth J. Goldman[1]
Department of Computer Science
Washington University
St. Louis, MO 63130
paul@cs.wustl.edu

1 Introduction

The Programmers' Playground is a software library and runtime system for creating distributed multimedia applications [1][2][3]. EUPHORIA[2] is the user interface management system for Playground, allowing end-users to create direct manipulation graphical user interfaces (GUIs) for distributed applications [5]. EUPHORIA has an intuitive graphics editor which allows end-users to simply draw GUIs (see Figure 1). The behavior of GUIs is established by forming logical connections between EUPHORIA and external Playground modules [4]. This document summaries the features of EUPHORIA and how it can be used to create GUIs. It is meant as a companion to the Playground reference manual [4].

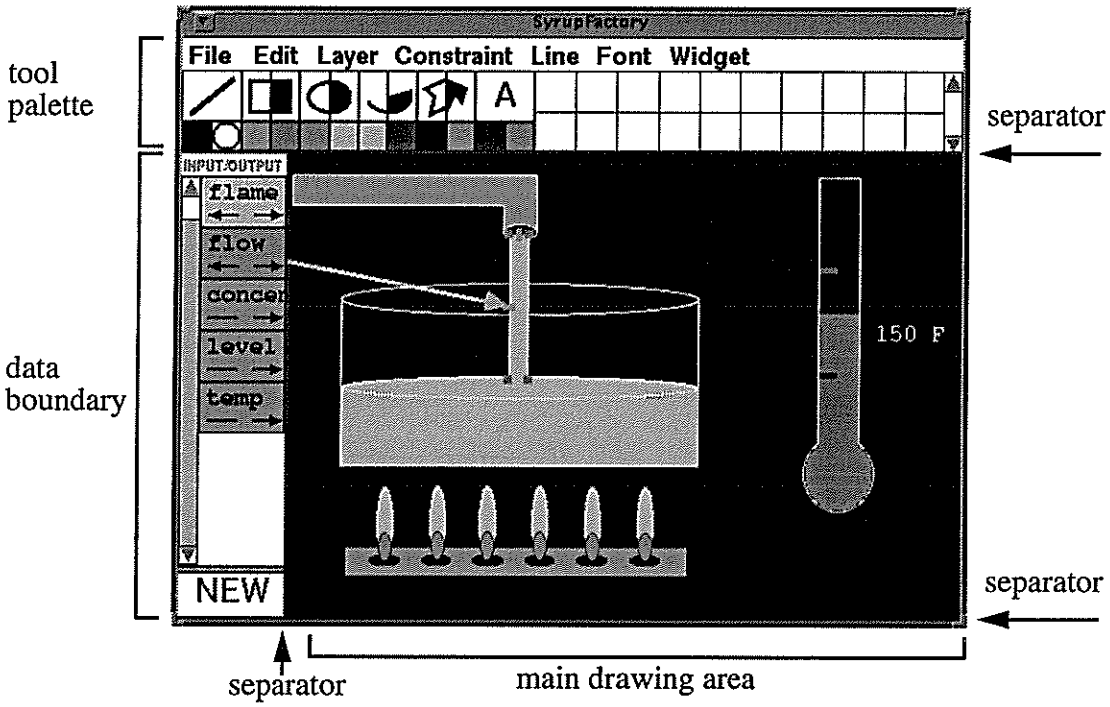

Figure 1: EUPHORIA graphics editor, displaying an interactive maple syrup factory GUI.

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.

---

1. This research was supported in part by National Science Foundation grants CCR-91-10029 and CCR-94-12711.
2. EUPHORIA is an acronym for End User Production of grapHical interfaces fOr Really Interactive distributed Applications.

1.1 Retractable panes

The EUPHORIA window is divided into a number of panes (see Figure 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.

Enlarging a pane can reveal additional information. For example, making the tool palette larger reveals a "background" label. Dragging a connection line from this to a color entry with the middle mouse button changes the background color of the main drawing area.

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 2).

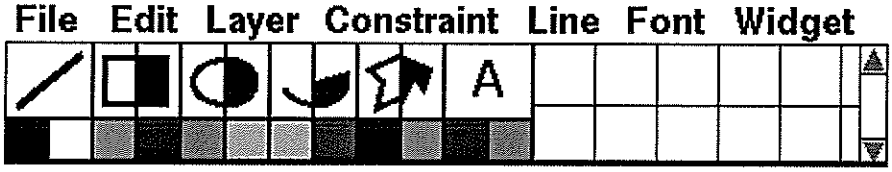

Figure 2: Tool palette.

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.

2.1 Selection & handles

In the main drawing area, clicking on a graphics object causes it to become selected or unselected. Multiple graphics objects an 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 over an area 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.

Figure 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 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 4.3).

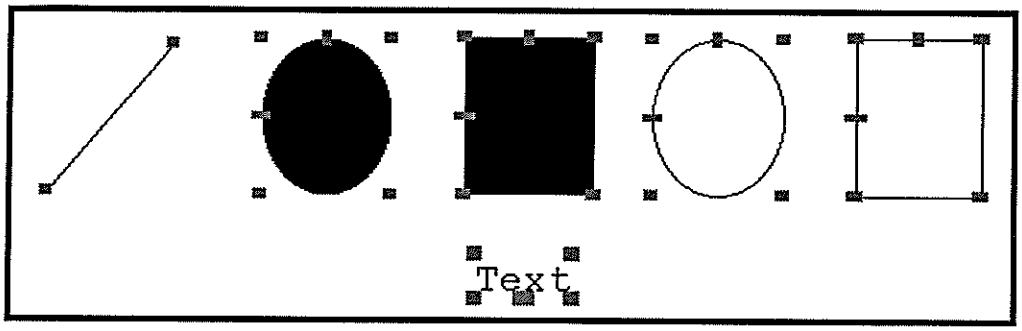

Figure 3: Selected graphics objects and their handles.

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.

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.

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 4). By

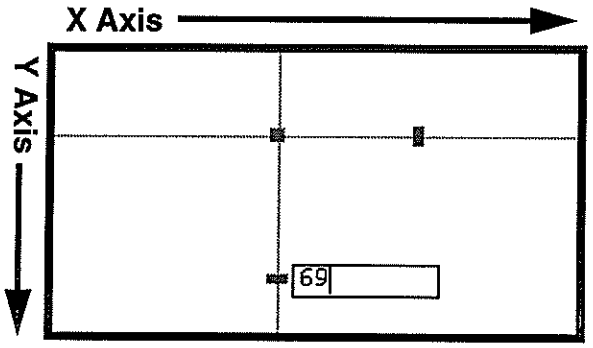

Figure 4: Coordinate system & origin controller.

default, the origin is at the top left corner of the main drawing area.

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 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 scaling factor of the axis is set by the system inferring that the distance from the origin to the selected position represents the entered coordinate value. For example, setting the value to a high number increases the scaling factor, making everything smaller.

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.

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 applications. 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 [1], each Playground module (including EUPHORIA) has a set of "published" externally readable/writable variables called the *data boundary*[3]. The data boundary is essentially an interface to the outside world.

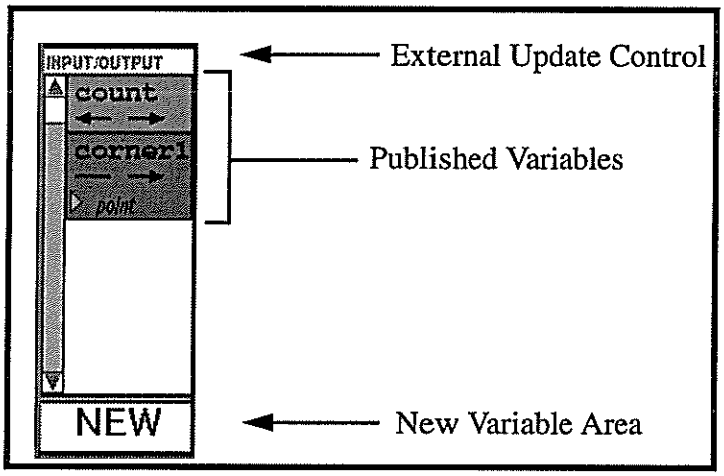

Figure 5: Data boundary.

Figure 5 shows the graphical representation of the data boundary which 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[4].

3.1 Published variables

A published variable represents a value which 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 [4].

Figure 6 shows the visual appearance of different types of published variables. As with handles, the color of a published variable represents its data type. Each variable has *protections*, represented as arrows, which control the read/write permissions of the variable to external modules [4]. 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.

3. In other publications, the term "presentation" has been used. The term "data boundary" is used to avoid confusion with graphical presentations.
4. It is possible to edit animated graphics objects with communication activated. However, it is sometimes hard to grab a quickly moving object!

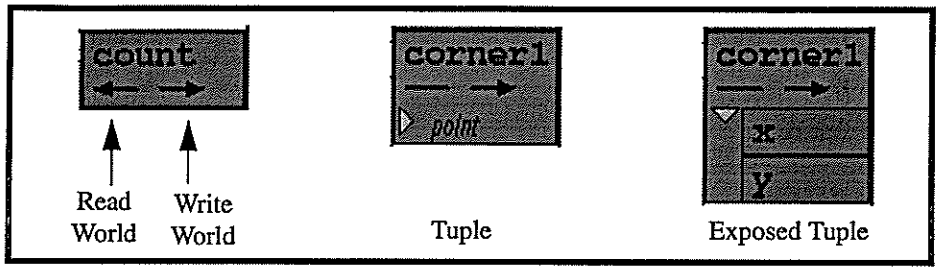

Figure 6: Published variables.

Clicking on a published variable selects/unselects it. Pressing the backspace or delete key removes the selected variables from the data boundary[5]. Double clicking on the name of a variable allows the name to be edited.

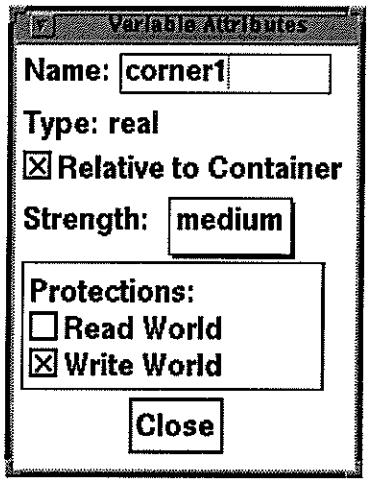

Figure 7: Variable attributes window.

Double clicking on a variable opens a dialog box for viewing and changing its properties (see Figure 7). Users can change the strength of a variable, which affects how the system interprets updates from external applications. For example, by default, user actions such as dragging a graphics shape take precedence over updates from a published variable (which is connected to the graphics shape). If the strength of the variable is set high enough, updates from the variable would 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.

3.2 Publishing a graphics attribute

A graphics object handle can be published, meaning that the attribute which it controls is connected to an externally readable/writable published variable. This is achieved by dragging, with the *middle mouse button*, a connection line from a graphics object handle and 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 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 4.2.

5. 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.

3.3 Creating user defined types

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.

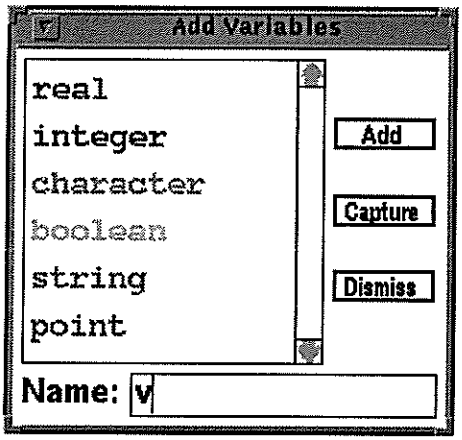

Figure 8: Add variables window.

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

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 6).

3.4 Declaring tuples in external applications

Point tuple variables and user defined tuples can be declared in external applications as described in [4]. For example, a point type can be declared in the following way:

```
PGtuple(Point) {
   PGreal x, y;
   PUBLIC_FIELDS(Point, field(x); field(y))
};
```

4 Constraints

Users can establish constraint relationships among the graphics object attributes, published variables, and individual fields of a tuple. 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. Three types of constraints are currently supported: constant, equality, and conversion.

4.1 Constant 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 the width and height handles of a rectangle are set to be constant, the size of the rectangle becomes fixed. A handle which is constant is colored gray. Clicking with the right mouse button on the handle a second time releases the constant constraint.

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 3.2 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 which 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.

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 usually 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 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 between modules.

Table 1: Supported equality (E) and conversion (C) constraints.

	real	integer	boolean	character	string	tuple
real	E	C			C	
integer	C	E			C	
boolean			E		C	
character				E	C	
string	C	C	C	C	E	
tuple						E/C

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.

4.4 Constraint visualization & editing

Constraints can be visualized and edited. In the "Constraint" menu, choosing **Show Constraints** enables constraint visualization. By default, constraints are shown as flashing lines between the handles of selected objects and/or published variables. A visualized constraint may be deleted by clicking on it with the *right mouse button*. Choosing **Show Current Propagation** visualizes the current computation direction [6] using an arrow head. Unsatisfied constraints are shown as dashed lines without an arrow head. Note that a visualization line or 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 5.1), and constraints in which at least one endpoint is not visible within the window. Choosing **Show Hidden Constraints** will show all constraints.

5 Advanced Drawing

EUPHORIA supports a number of high level mechanisms for constructing GUIs, including imaginary objects, alternatives, widgets, and aggregate mappings (under development).

5.1 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.

For example, ovals in EUPHORIA do not have handles in their center. However, through the use of two imaginary rectangles and some constraints, it is possible to create a oval center handle (see Figure 9). The sizes of the two rectangles are constrained to be equal and the bottom-right corner of one rectangle is constrained to be equal to the top-left of the other. These rectangles are then inscribed within the oval. The result is a handle which is always in the center of the oval, even if the oval is moved or resized.

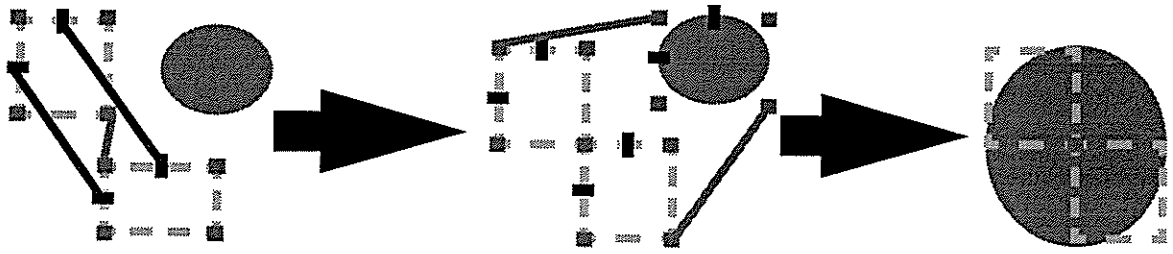

Figure 9: Creating a center handle for an oval from imaginary graphics objects.

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

5.2 Alternatives

A user GUI can have multiple representations which are called *alternatives*. For example, a simulation GUI might consist of an alternative which shows the simulation state graphically, allowing direct manipulation, and an alternative that shows expanded information in a more "text and button" type representation. Alternatives are often used in the development of widgets (see Section 5.3). For example, a widget may have standard and selected visual representations which are defined as separate alternatives.

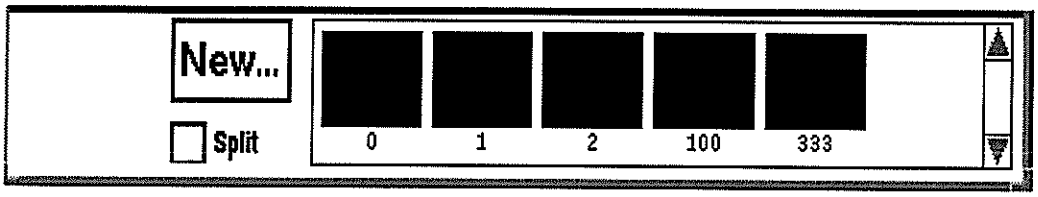

Figure 10: Alternatives interface.

At the bottom of the EUPHORIA window is a hidden pane for specifying alternatives; this pane can be exposed by dragging up the bottom divider (see Figure 10). A table lists each alternative as a box with an associated alternative ID. Clicking on an alternative displays it in the main drawing area (only one alternative is displayed at a time). Any drawing in the main drawing area becomes incorporated into the current alternative. Initially, there is only one alternative, with ID 0. Pressing the **New...** button creates a new alternative based on a supplied alternative ID.

5.3 Widgets

A widget is a compound graphics object that is a grouping of graphics objects with a subset of exposed attributes. One may think of a widget as a "module" of graphics shapes with a data boundary of externally readable/writable attributes. The values of the attributes in a widget's data boundary are the only means of controlling or viewing the state of the widget externally. Widgets are created visually by end-users. As with other graphics objects, the external attributes of a widget can be viewed, revealing handles which can be used in forming connections to the widget.

Users can create a widget in the following way. First, a drawing is made in the main drawing area as described in Section 2. Note that this can also include constraint relationships among graphics objects. Second, the attributes of the graphics shapes in the drawing which are to be exposed, are published as described in Section 3.2. All other graphics attributes will be essentially encapsulated within the widget. Third, the drawing is saved to a file. Finally, choosing **Load As Widget...** from the "Widget" menu creates a widget from the saved specification. The graphics objects of the drawing are grouped within the widget; the published attributes appear as handles.

The graphics objects within a widget are in a separate coordinate system (see Section 2.4). In publishing attributes of a widget, users can specify how the exposed handles of a widget translate values coming into and going out of the widget. Values of the widget's handles can be in terms of the widget's local coordinate system, or the coordinate system of its container (i.e. the main drawing area, or another widget which the widget is contained within). This translation is set in the *Variable Attributes Window* described in Section 3.1.

6 Command Line Arguments

A number of optional command line arguments allow users to customize the execution of EUPHORIA:

Table 2: Optional command line arguments.

argument	default	description
`-buffer`	290x400	Size of offscreen buffer, used for screen updates.
`-colorDelta`	200	Maximum color approximation distance in RGB space.
`-display`	no default	X windows display name [8].
`-file`	no default	Saved EUPHORIA file to load on start-up.
`-geometry`	580x400-0+0	Position and size of the EUPHORIA window [8].
`-invalidAreas`	3	Number of invalid rectangles maintained.
`-pollDuration`	50	Event polling time before drawing, in msecs.
`-pollSleep`	10	Sleep time while polling for events, in msecs.
`-title`	EUPHORIA	Title of EUPHORIA window and module name.

For example, to start EUPHORIA with specific display and a small buffer:

```
PGeuphoria -display seesaw.cs.wustl.edu:0.0 -buffer 200x200
```

6.1 Double buffering

Double buffering is used for smooth, flicker free, graphics rendering. This means that a resource called a "pixmap" must be allocated to buffer intermediate drawing results. The size of the pixmap is determined by the `-buffer` argument. Setting this value to a large size can result in more efficient drawing. Unfortunately, large pixmaps use a lot of memory; setting this value too large can cause EUPHORIA not to start, giving an X-windows warning.

6.2 Color allocation

Workstations which have a limited number of colors (e.g., 8 bit depth or 256 simultaneous colors) can have problems managing how colors are allocated. EUPHORIA controls how color is allocated, and can approximate a requested color to an already allocated color. Color approximation degree is set by the `-colorDelta` option. Color delta is the maximum distance in RGB space in which two colors can be considered equivalent. Setting this value lower will tend to match the requested values more exactly (e.g., setting color delta to 0 disables color approximation).

6.3 Invalidation

Multiple "invalid areas" can be maintained for the EUPHORIA window. These areas determine which portions of the window need to be redrawn when the appearance of window items change. Having more invalid areas is likely to make drawing more efficient if the buffer is small

or many sparsely positioned, disconnected graphics items change sporadically. On a workstation with fast graphics capabilities, fewer invalid areas may result in more efficient drawing.

6.4 Event loop

EUPHORIA's event loop is timed according to the `-pollSleep` and `-pollDuration` arguments. Before drawing is performed in an iteration of the event loop, the system first polls for events and updates from the Playground environment. The polling time is determined by poll duration. This allows the system to gather many changes to draw simultaneously, rather than drawing each change separately. The duration effectively determines the maximum "frames per second" update rate of the drawing. The default setting allows for at most 20 updates per second; setting this value higher can result in more efficient, but "jumpier", drawing. During the polling loop, EUPHORIA repeatedly sleeps for a period of time (determined by loop delay) to wait for new events and to give other processes a chance to run. Setting value this lower can result in faster drawing. However, this can cause EUPHORIA to monopolize the CPU of the workstation on which it is running.

7 Common Questions

Q: *Why do graphics objects sometimes change shape during dragging or external updates?*

A: EUPHORIA uses a constraint solver not only for end-user constraints, but also for direct manipulation and external updates. A set of constraints can be *underconstrained*, causing these types of problems. This means that the constraint solver may make arbitrary choices on how to satisfy a set of constraints. This usually can be solved by adding more constraints. For example, adding constant constraints to the width and height of a shape.

Q: *Why does EUPHORIA occasionally ignore some of the constraints?.*

A: In general, it is not always possible to solve all constraint relationships. A set of constraints can be *overconstrained* if two or more constraints conflict with each other. In the event of conflicts, one or more constraints may be left unsatisfied. Also, cyclic relationships of constraints may cause constraints to be unsatisfied. Unsatisfied constraints are shown as dashed lines in when visualized (see Section 4.4). The solution to overconstrained constraints is to simplify the constraint relationships between graphics objects.

Q: *Why is the EUPHORIA module sometimes not removed from the connection manager user interface after EUPHORIA is exited?*

A: Playground's connection manager is unable to detect when a module has been terminated through the use of the UNIX "kill" command. If EUPHORIA is exited using "kill", then the module with continue to appear in the connection manager user interface. Note that this includes the window system methods of quitting a window, such as choosing **Quit** from the OpenWindows default window menu. The correct way to exit EUPHORIA is to choose **Quit** from EUPHORIA's file menu.

Q: *How can I make EUPHORIA run faster?*

A: Table 2 lists a number of options for fine tuning the execution of EUPHORIA. In designing a GUI, one should take into account the speed of the hardware on which it is run and the user perception of change. That is, attempting to update a GUI at faster rate than the hardware can handle or faster than a user can perceive, can result in a GUI that runs slowly. It must be remem-

bered that EUPHORIA is part of a distributed system; if the EUPHORIA module monopolizes the CPU, external modules will run slower which, in turn, will make EUPHORIA run slowly. Also, if other modules update their variables repeatedly at a rate faster than the update rate of EUPHORIA, time is still spent dealing with the intermediate values of the variables even though the values may be "skipped over". External modules should utilize Playground's `PGsleep` [4] to adjust their speed to a reasonable rate, and should avoid resending redundant information.

Q: *I'm updating the position of an object on the screen from an external module. Why does the object "hop", first moving on the x-axis and then on the y-axis?*

A: The problem is that the external module should be using Playground's atomic step mechanism [4]. This ensures that changes to the x coordinate and the y coordinate occur together. Another way to make drawing more efficient is to have all changes for an iteration within an atomic step, eliminating redundant drawing.

Acknowledgments

We thank the EUPHORIA users of the Washington University CS333 class for their useful comments. We also thank David Saff, who is in the process of developing constraint visualization and editing for EUPHORIA. We thank Bala Swaminathan and Ram Sethuraman for their work in developing the Playground library.

References

[1] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 363-372, January 1994. Long version available as Washington University Department of Computer Science technical report WUCS-93-29.

[2] Kenneth J. Goldman, T. Paul McCartney, Bala Swaminathan, and Ram Sethuraman. The Programmers' Playground: A demonstration. In Proceedings of the 1995 ACM International Conference on Multimedia, October 1995. To appear.

[3] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications. IEEE Transactions on Software Engineering. To appear.

[4] Kenneth J. Goldman, T. Paul McCartney, Ram Sethuraman, Bala Swaminathan, and Todd Rodgers. Building Interactive Distributed Applications in C++ with The Programmers' Playground. Washington University Department of Computer Science technical report WUCS-95-20.

[5] T. Paul McCartney and Kenneth J. Goldman. Visual Specification of Interprocess and Intraprocess Communication. In Proceedings of the 10th International Symposium on Visual Languages, pages 80-87, October 1994.

[6] T. Paul McCartney. UltraBlue: A Cycle Breaking Multi-way Constraint Solver. Washington University Department of Computer Science technical report (in preparation).

[7] Playground's World Wide Web page: **http://www.cs.wustl.edu/cs/playground**

[8] Robert W. Scheifler, James Gettys. X Window System, Third Edition. Digital Press, 1992.