EUPHORIA Reference Manual

T. Paul McCartney, Kenneth J. Goldman

WUCS-97-13

February 1997

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

*Revised for **EUPHORIA v3.13***
February 1997

Earlier version published as
Washington University technical report WUCS-95-19.

Abstract

EUPHORIA is a user interface management system that enables end-users to create direct manipulation graphical user interfaces (GUIs) through interactive drawing. Used in conjunction with The Programmers' Playground, a distributed programming environment, end-users can dynamically create and associate GUI components with an underlying application without programming. This document describes EUPHORIA's functionality.

Table of Contents

1   Introduction

The Programmers' Playground is a software library and run-time system for creating distributed multimedia applications [1], [2]. Playground is based on *I/O abstraction*, a programming model for distributed systems that provides a separation of communication and computation. A distributed application consists of a set of communicating *modules*, written as C++ programs, using special data types from the Playground library. Variables declared using these data types can be *published*, making information available to external modules. The communication structure among the modules of a Playground application is defined by a set of *logical connections* among published variables of the modules. Communication among modules of a distributed system occurs implicitly; when a published variable is modified, the new value of the variable is automatically sent to its connected variables. In this way, each module can be created independently of the modules with which it communicates.

This document summarizes the features of EUPHORIA, a user interface management system for Playground. With EUPHORIA, end-users can create direct manipulation graphical user interfaces (GUIS) interactively through the use of an intuitive graphics editor. EUPHORIA, implemented as a Playground module, can be connected to other modules of a distributed application through the use of a visual configuration language [3]. This has the effect of associating an end-user GUI with its underlying application.

- Updates to published variables in external modules can change the properties of the end-user drawing in EUPHORIA.

- Manipulation of graphics components in EUPHORIA can change published variables, sending changes to external modules.

Figure 1: External modules configured to EUPHORIA.
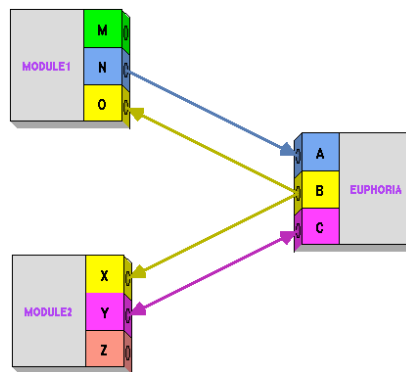
For example, Figure 1 shows three modules configured to form a distributed application: MODULE1, MODULE2, and EUPHORIA. Whenever variable N or Y is changed in MODULE1 or MODULE2, the changes are sent to EUPHORIA, possibly resulting in animation within the end-user's GUI. Similarly, whenever variable B or C is modified in EUPHORIA, these changes are sent to MODULE1 and MODULE2.

2   Basic Drawing

EUPHORIA's graphics editor consists of four parts (Figure 2): tool palette, data boundary, main drawing area, and alternatives palette (not shown). Drawing in EUPHORIA is much like drawing in other graphics editors such as MacDraw or FrameMaker. With the tool palette, users can select shapes to be drawn, change the color of shapes, and perform other operations through the menus.
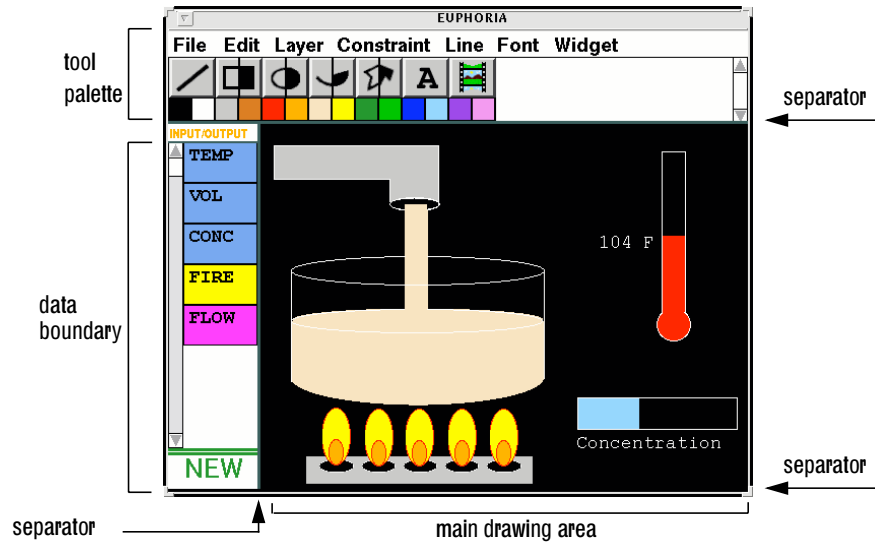
Figure 2: EUPHORIA graphics editor, displaying an interactive maple syrup factory GUI.

Drawing shapes

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 causes it to remain selected after the first drawing. In this way, users can conveniently draw multiple shapes. When finished, clicking on a selected tool unselects it.

<u>*NOTE:*</u>   EUPHORIA *is meant to be controlled by a three button mouse. Throughout this document, whenever the words "click" or "drag" are used, the left mouse button is implied.*

To change the color of shapes or background

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 shapes 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 main drawing area.

To hide the tool palette and data boundary

The EUPHORIA window is divided into a number of panes (see Figure 2). 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. When a GUI is saved/loaded, the associated pane sizes are also saved/loaded.

2.1   Selection & Handles

In the main drawing area, clicking on a graphics object causes it to become selected. When no drawing tools are selected, dragging a selection box in the main drawing area will select all graphics shapes within the box. All previously selected graphics shapes become unselected when dragging a selection box.

- Clicking on a selected shape again causes it to become unselected.

- Multiple graphics shapes can be selected at the same time; selecting a graphics object does not unselect other graphics shapes.

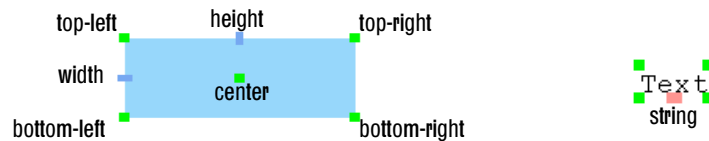- Clicking in the main drawing area's background causes all selected shapes to become unselected.

Figure 3: Selected graphics shapes and their handles.

Figure 3 shows the selection handles of rectangle and text graphics shapes. As with other graphics editors, most of these handles can be dragged to change the attributes of their graphics shape. 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 shapes, 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).

To delete a shape

Pressing the backspace or delete key will delete all selected shapes in the main drawing area. *Currently there is no way to undo a delete operation.*

2.2   Layering

To control the order in which graphics shapes are drawn, each shape has an associated layer attribute. When a shape is first created, it is set to the front-most layer. This means that it is to be drawn on top of all other shapes. A selected shape's layer can be changed with the **Layer** menu, by choosing either **Bring to Front** or **Send to Back**.

2.3   Coordinate System

The main drawing area's coordinate system is oriented with the x-coordinate axis increasing to the right, and the y-coordinate axis increasing downward. By default, the origin is at the top-left corner of the main drawing area. An *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. Users can change the position of the origin and the scaling factor of the x and y axes.
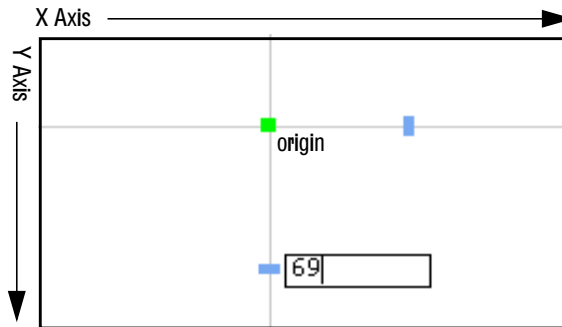
Figure 4: Coordinate system and origin controller.

To change the origin or scale of a drawing

The origin controller (see Figure 4) 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 shapes in the main drawing area shorter.

3   Data Boundary

EUPHORIA's set of published variables is known collectively as the *data boundary* (also known as the "presentation"). As described in Section 1, EUPHORIA's published variables can be used to establish communication between a drawing's components and external distributed application modules. This allows external modules to both modify graphics shapes and to receive updates from end-user direct manipulation. As a result, interactive, animated graphical displays may be created by the end-user. Figure 5 shows the graphical representation of the data boundary that appears as the left portion of the EUPHORIA window.

Figure 5: Data boundary.

To disable/enable external communication

The top portion of the data boundary contains an "External Update Control" button. Clicking on this button toggles between **Input/Output** and **Frozen** modes, allowing users to enable or disable communication between EUPHORIA and external modules. It is sometimes useful to turn off communication for a period of time in order to make it easier to modify graphics shapes.

3.1   Published Variables

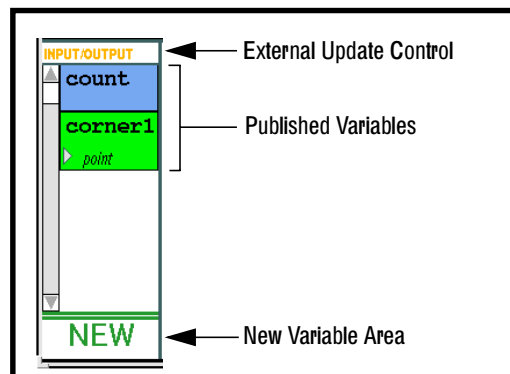A published variable represents a value that is shared with external Playground modules. When a variable is changed in an external module, Playground sends the change out to all connected modules, including EUPHORIA. Similarly, when a graphics shape is modified (e.g. moved by the user), this change may also be sent out to external Playground modules, according to the published variables and the logical connections between variables (see Section 1). Figure 6 shows the visual appearance of a few different types of published variables. As with handles, the color of a published variable represents its data type.
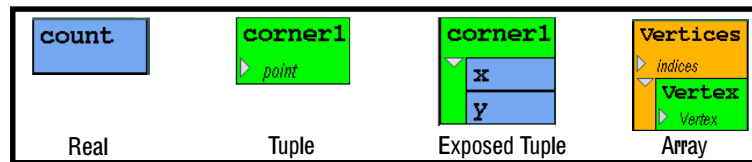
Figure 6: Published variables.

To publish a graphics attribute

A graphics shape 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 the Playground environment of the variable, and forming a constraint (see Section 4) between the handle's graphics shape attribute and the published variable.

To delete or rename a variable

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

⚠ *Pressing backspace deletes both selected variables and selected graphics shapes. Be careful that shapes are not selected before deleting a variable.*

3.2   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 name, strength, protections, and other variable attributes.

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
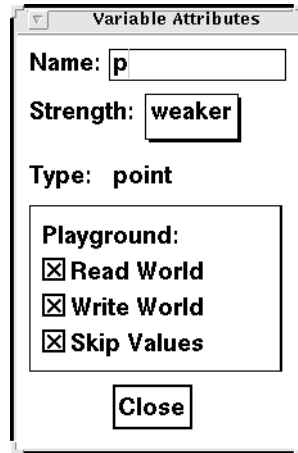
Figure 7: Variable attributes window.

the graphics shape). If the variable's strength is set sufficiently high in relation to other system strengths (Section 4.5), updates from the variable take precedence over user actions.

Each variable has *protections* that control the read/write permissions of the variable to external modules [3]. 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.

3.3   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.

Figure 8: Add variables window.

User defined tuple data types can be created that consist of multiple fields of data, including other tuples. User defined arrays of tuple data types may also be created.

To create a user-defined tuple type

Choosing **Capture Tuple Type** from the **Types** menu of the Add Variables window creates a new tuple type using the currently displayed data boundary variables as the tuple fields and the entered name as the type name. This new type is inserted into the list of available types.

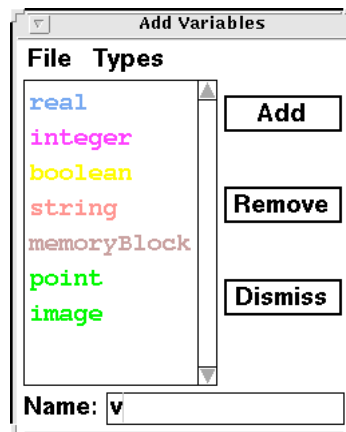Variables of a user defined tuple can be published by pressing the **Add** button just as with other types. When published into the data boundary, a tuple appears in green with a small triangle to the left. This triangle is used to expose or hide the fields of the tuple (see Figure 6).

To create a user-defined array of tuples type

Choosing **Create Array Type...** from the **Types** menu of the Add Variables window brings up a window for creating static array types. Within this window, one can specify the tuple element type and array size. Types created this way are added to the Add Variables window list of types.

Variables of a user defined array can be published by pressing the **Add** button just as with other types. When published into the data boundary, an array appears in orange with two small triangles to the left. These triangles are used to expose or hide the dimensions and field type of the array (see Figure 6).

4   Constraints

A constraint is a persistent relationship to be maintained among graphics shape attributes.   Users can establish constraints among the graphics shapes attributes and/or published variables. Once a constraint is formed, the system is responsible for maintaining the relationship when changes are made to graphics shapes or published variables. For example, one could constrain the width and height of a rectangle to be equal, constraining it to be a square; manipulating the height of the rectangle causes the width to also change, maintaining the constraint relationship.

Four types of constraints are supported: *anchor*, *equality*, *conversion*, and *formula*.

4.1   Anchor Constraints

An anchor constraint is used to set a graphics shape attribute constant, so that it cannot be changed accidentally. For example, one can anchor the top-left and bottom-right of a rectangle to prevent it from being moved.

☞   Clicking on a handle with the *right mouse button* causes the corresponding attribute of the graphics shape to become anchored. Clicking with the right mouse button on the handle a second time releases the anchor constraint.

4.2 Equality Constraints

☞ An equality constraint can be established by dragging a connection line between two graphics shape handles of the same type with the *middle mouse button*.
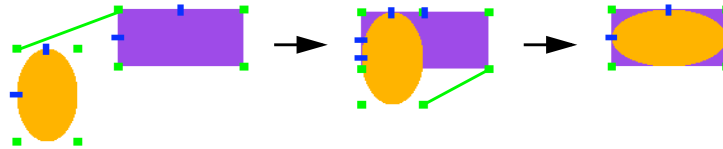
Figure 9: Inscribing an oval within a rectangle.

For example, an oval can be inscribed within a rectangle through the use of two equality constraints (see Figure 9). A constraint is formed between the top-left handles of the oval and rectangle, causing the shapes to "snap together" (established constraints are shown here as lines). A second constraint is formed between the bottom-right handles, resulting in the oval changing shape to fit into the rectangle. Since these relationships are persistent, resizing or moving either of the shapes causes the other to also change.

Equality constraints can also be formed between graphics object handles and published variables. 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.

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 1 lists the supported connection types. Note that only a subset of these conversion operations are available for making connections among Playground modules.

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

	real	integer	boolean	string	memory Block	tuple
real	E	C		C		
integer	C	E		C		
boolean			E	C		
string	C	C	C	E		
memory Block					E	
tuple						E/C

- Conversion 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   Formula Constraints

With a calculator object one can specify a constraint relationship among graphics shapes in terms of an arbitrary algebraic formula. A multi-way constraint graph [4] 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 5.2).
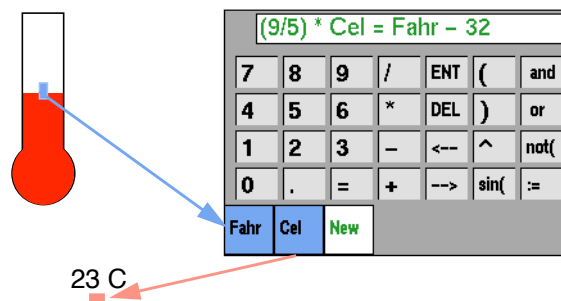

Figure 10: A calculator object for converting between temperature scales.

For example, a calculator object could be used to convert between scales of measurement, such as Celsius and Fahrenheit temperatures (see Figure 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.

To create a formula constraint

Selecting **New Calculator** from the **Constraints** menu creates a new calculator in the main drawing area. Dragging from graphics shape handles with the middle mouse button to the calculator's "New" area (see Figure 10) creates both calculator variables and equality constraints between the variables and the corresponding graphics shape attributes. A formula constraint is created by entering an algebraic equation in terms of the calculator variables and pressing Return.

- Clicking in the calculator's **New** area reveals a pop-up menu that can be used to create variables.

- A variable may be renamed by double-clicking on its name.

4.5   Constraint Strengths

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.
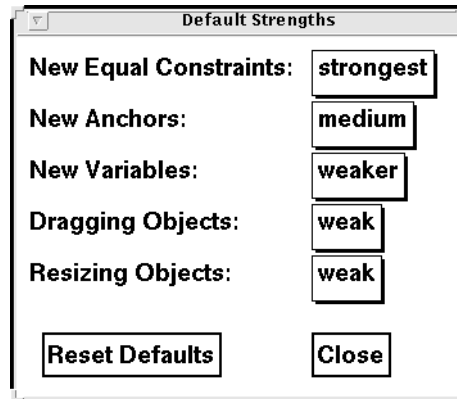
Figure 11: Strengths window

Figure 11 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 shapes. By changing the strengths on these operations results in different interactive behavior in EUPHORIA.

For example, by default, anchor constraints are stronger than variable and dragging constraints. With these settings, if a shape's position is anchored then the position cannot be changed by dragging or 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 shape's position but user dragging would still not affect the position.

4.6   Constraint Visualization and Editing

Constraints can be visualized and edited. In the **Constraint** menu, choosing **Show Constraints** enables constraint visualization of selected graphics shapes and published variables.
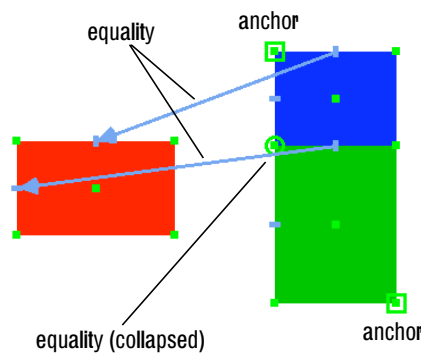
Figure 12: Constraint visualization.

Figure 11 shows an example of constraint visualization with three rectangles. Equality constraints are shown as flashing arrows between the handles of selected objects and/or published variables. The direction of the arrow represents the constraint's computation direction (i.e., which attributes are computed from which other attributes).   When handles overlap, the visualization of an equality constraint is collapsed to a circle around the corresponding handles. Anchor constraints are shown as squares.

- Unsatisfied constraints are shown as dashed shapes.

- A visualization arrow can represent multiple constraints, in the case of tuples. Double headed arrows are used to show the mixed computation directions.

To edit constraints

☞ 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.

To view hidden constraints

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

To view collapsed and "crowded" 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 shapes. In this mode, graphics shapes 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 shapes 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.

5   Advanced Drawing

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

5.1   Movies

The movie tool in the toolbar (Figure 13) represents the creation of a movie graphics shape. A movie consists of a series of numbered *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 to the data boundary as an image tuple (Section 7), allowing external modules to send frames to the movie.

When a new image value is received by a movie, the image's "ID" field is used to determine whether a new movie frame should be created or to replace an existing movie frame (i.e., if an image arrives at the movie with ID=x, it will replace frame #x if it exists). The movie's frame # handle is used to specify the ID of a movie frame to display.
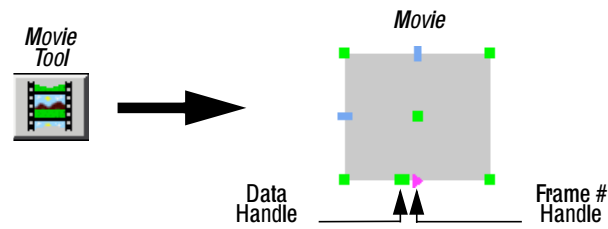
Figure 13: Movies

⚠ *It is usually necessary to disable the "Skip Values" attribute of the image published variable associated with a movie (see* Section 3.2*). Otherwise, some image frames may be skipped and will not arrive at the movie.*

5.2   Imaginary Objects

Constraints are an intuitive way of establishing direct relationships (e.g., equality) among graphical attributes. However, many times it is necessary to create indirect relationships among graphics attributes where there are one or more intermediate computations involved in relating the attributes. EUPHORIA's *imaginary objects* mechanism allows end-users to define relationships among graphics shapes through the use of intermediate, invisible graphics shapes, serving as an abstraction for defining indirect constraint relationships.
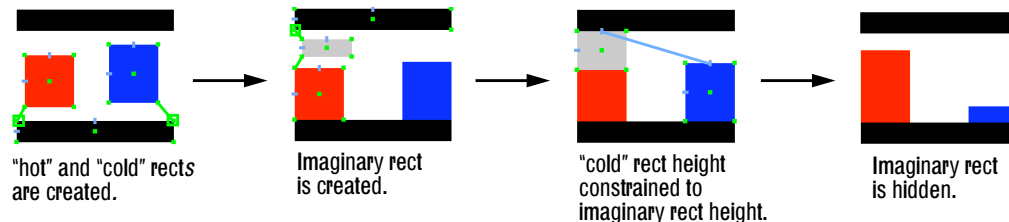

Figure 14: Using imaginary objects to construct a temperature controller.

For example, an imaginary object can be used to create a temperature controller (Figure 14) that displays the proportion of "hot" (shown as a red rectangle, left) to "cold" (shown as a blue rectangle, right). The desired behavior is that the red rectangle height should move inversely to the blue rectangle's height (e.g., dragging the red rectangle taller should result in the blue rectangle becoming shorter). Figure 14 shows the stages in creating the controller. First, the red and blue rectangles are created and anchored to the bottom portion of the controller (established equality constraints are shown here as lines; anchors are shown as squares). Second, an imaginary rectangle is created and is constrained to be positioned between the red rectangle's top and the controller's top. Third, the height of the blue rectangle is constrained to be equal to the imaginary rectangle's height. Finally, the imaginary rectangle is hidden. Manipulating either the red or blue rectangles' height has the effect of changing the other's height through the imaginary object.

Using imaginary objects

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

5.3   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 5.4).
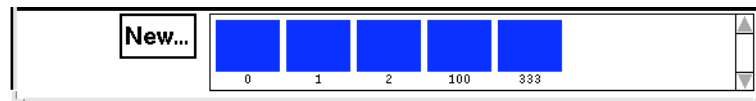

Figure 15: Alternatives pane.

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 15). A table lists each alternative as a box with an associated alternative ID. Clicking on an alternative box causes it to become selected, displaying its contents in the main drawing area (only one alternative is displayed at a time). Any drawing in the main drawing area becomes incorporated into the currently selected alternative. Initially, there is only one alternative, with ID = 0. Pressing the **New...** button creates a new alternative based on a supplied alternative ID.

5.4   Widgets

A widget is an encapsulated grouping of graphics shapes with a data boundary of exposed attributes. The attribute values in a widget's data boundary are the only means of controlling or viewing the state of the widget externally. As with other graphics shapes, the external attributes of a widget can be viewed as handles which can be used in forming connections to the widget. The graphics shapes within a widget are in a separate coordinate system (see Section 2.3), allowing the widget to be defined in terms of meaningful, real-world values rather than actual raw pixel values.

To define a widget

First, a drawing is made in the main drawing area as described in Section 2. This can also include constraint relationships among graphics shapes. Second, the graphics shape attributes in the drawing that are to be exposed, are published as described in Section 3.3. All other graphics attributes will be encapsulated within the widget. Third, the drawing is saved to a file.

To use a widget

Choosing **Load As Widget...** from the **Widget** menu creates a widget from the saved specification. The graphics shapes of the drawing are grouped within the widget; the published attributes appear as handles. The widget has a number of default handles for controlling its attributes, such as width and height. Also, each widget has an alternative ID handle that is used to control which widget alternative is currently being viewed.

For example, a thermometer widget can be constructed as follows. First, the component shapes of the thermometer are drawn and constraints among the shapes are formed (Figure 16a). A scaling factor for the widget space is set using the Origin Controller (Section 2.3) so that the top of the thermometer represents 300 degrees Fahrenheit. This allows external applications to interact with the thermometer in terms of real world values rather raw pixels. Second, the data boundary of the widget is defined by publishing the height of the mercury (i.e., the temperature, Figure 16b). The data
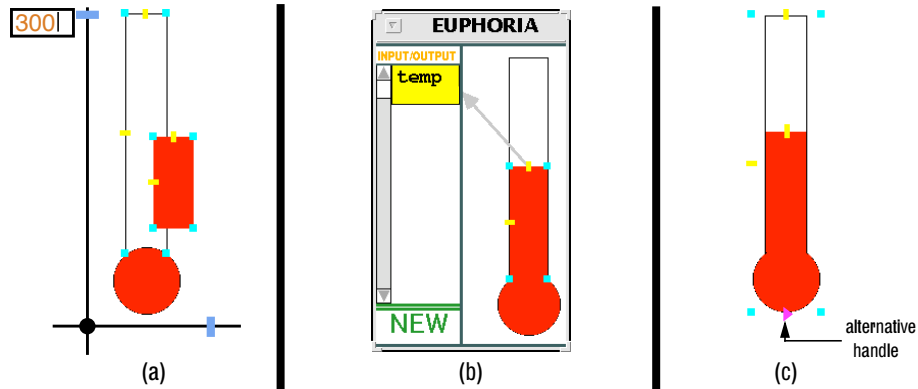
Figure 16: Creating a thermometer widget.

boundary specifies that only the temperature attribute will be exposed from the widget when it is used. The specification of the widget is saved, and the widget can then be used within a GUI (Figure 16c).

5.5   Aggregate Mappings

An *aggregate mapping* is a mechanism for visualizing and manipulating the elements of an aggregate. EUPHORIA supports aggregate mappings of static arrays (see Section 3.3 for instructions for creating array types). End-users can define an aggregate mapping by specifying a single graphics shape, called the *prototype instance* and its relationship (i.e., constraints) to the aggregate's *element type*. A copy of the prototype instance is created for each aggregate element according to the specified relationship, resulting in an interactive visualization of each aggregate element. The prototype instance becomes imaginary after the aggregate mapping is established.

To define an aggregate mapping

Given an array in the data boundary (Section 3.3), expose the array's element type by clicking on the triangle pointing to the element type label. Create a prototype instance that will represent each aggregate element either by drawing a simple shape or loading a widget (Section 5.4). Form constraints between the array's element type fields and the prototype instance. When finished, select both the array and the prototype instance and choose **Define Aggregate Mapping** from the **Edit** menu.
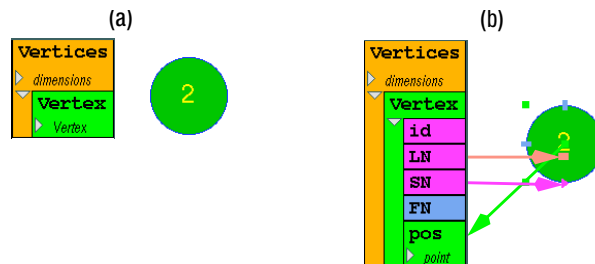

Figure 17: Aggregate mapping of a vertices array.

For example, the vertices of a graph can be visualized using an aggregate mapping. Figure 17a shows an array with a tuple representative element (i.e., Vertex) and a prototype instance (i.e., a vertex widget). In Figure 17b, the end-user defines constraints among the fields of the representa-

tive element and the prototype instance. The `LN` field is connected to the widget's text handle, the
`SN` field is connected to the widget's alternative handle, and the `pos` field is connected to the wid-
get's center handle. Defining the aggregate mapping has the effect of creating a copy of the vertex
tuple for each element of the array, with constraints created between each array element and each
copy of the widget.

Filtered Aggregate Mappings

It is often is desirable to view only a subset of an aggregate's elements. Filtering out extraneous
elements results in a simplier display that is easier to comprehend. An *aggregate filter* mechanism
that allows the end-user to filter the displayed elements based on a predicate. For example, one
could specify the predicate of "FN < 100" on Figure 17's aggregate mapping, having the effect of
only displaying vertices whose FN field is less than 100.

Specifying an aggregate filter

Filtering is achieved by creating a *calculator object* (see Section 4.4) and making connections
among the representative element's fields and the calculator (this should be done before the aggre-
gate mapping is defined). The calculator should be selected along with the prototype and the aggre-
gate when **Define Aggregate Mapping** is initiated.

Joined Aggregate Mappings

Many times, single aggregates do not contain all of the relevant information needed to make a
desired aggregate mapping. Instead, information may be spread among multiple aggregates from
external sources. With a *joined aggregate mapping*, the data of multiple aggregates is coordinated
within an aggregate mapping based on a matching operation between *key fields* (currently, only
integer key fields are supported).

Joining aggregate fields

Join operations of an aggregate should be specified before the aggregate mapping is defined. Drag-
ging with the *middle mouse button* from a key field of the destination aggregate to a key field of the
source aggregate creates a virtual representation of the source aggregate's element type in place of
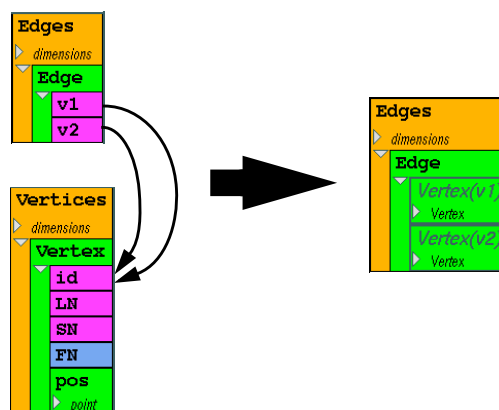the key field of the destination aggregate.

Figure 18: Joining Edges and Vertices aggregates.

For example, the graph visualization example uses an aggregate mapping to display the edges of the graph. To properly display an edge, it is necessary to know the positions of the edge's two associated end-points. However, the Edges aggregate does not explicitly store this information (see Figure 18). Instead, it stores a set of paired ID numbers v1 and v2 of the end-points; the vertex position information is stored separately in the Vertices aggregate. Joining the v1 and v2 key fields of Edges to the id key field of Vertices creates virtual representations of the Vertex tuple within the Edges aggregate that can be used in establishing an aggregate mapping of edges. Note that the Vertices aggregate still remains after forming the join, and can be used in forming a separate aggregate mapping.

6   Setup

6.1   Command Line Arguments

Optional command line arguments allow users to customize the execution of EUPHORIA.

Table 2: Optional command line arguments.

argument	default	description
-buffer	580x400	Size of offscreen buffer, used for screen updates.
-colorDelta	200	Maximum color approximation distance in RGB space.
-display	no default	X windows display name [5].
-file	no default	Saved EUPHORIA file to load on start-up.
-geometry	580x400-0+0	Position and size of the EUPHORIA window [5].
-invalidAreas	3	Number of invalid rectangles maintained.
-personalDir	home directory	Location of ".euphoria" directory to be created/used.
-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.
-url	no default	WWW address of a EUPHORIA file to load.

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

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

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 due to lack of memory, giving an X-windows error.

Color Allocation

Workstations that 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).

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.

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 workstation's CPU.

6.2   Personal Directory

EUPHORIA uses a "personal directory" to store preferences and other information. The `-personalDir` command line argument is used to specify where the personal directory is located. If the personal directory cannot be found, EUPHORIA creates one automatically in the user's home directory called ".euphoria." Currently, two items are used from the personal directory: a `euphoria_home` file and a `download` directory.

☞ The `euphoria_home` file should contain a single line with the full pathname of the main EUPHORIA directory. The main EUPHORIA directory contains shared files used by multiple users. For example, the tool icons of Figure 2 are loaded from this directory; if you do not set the euphoria_home file with the correct path, you will use an alternative tool palette.

The `download` directory is used to store files which are loaded from the World Wide Web with the **Load URL...** file command or the `-url` command line option.

7   Predefined Types

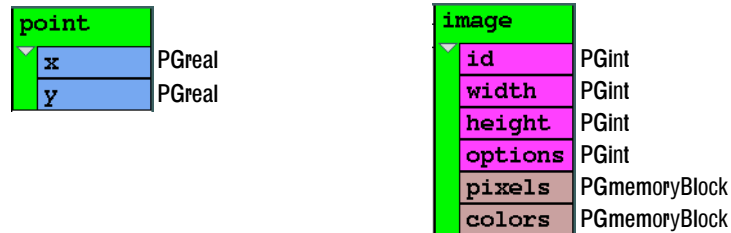Currently, EUPHORIA supports two predefined compound data types: *point* and *image*.

Figure 19: Predefined tuple types.

The point tuple represents an (`x`, `y`) Cartesian coordinate.

A PGimage C++ class is available that implements the image tuple.

The image tuple represents a single image with ID = `id` (see Section 5.1) and dimensions of `width` by `height` pixels. The `options` field is currently unused; for upward compatibility, modules should set this field to 0. The `pixels` field stores the image pixels in uncompressed in "row-major" order, using one byte per pixel. Each pixel value is an index into a color table stored as the `colors` field. Given a color index n, the color value is stored in `colors` at byte indices 3*n (red), 3*n + 1 (green), and 3*n + 2 (blue).

8   Common Questions

Q: *Why do graphics shapes 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. Usually this can be solved by adding more constraints. For example, adding anchor 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 4.6). The solution to overconstrained constraints is to simplify the constraint relationships between graphics shapes or to change the strengths of some of the constraints (Section 4.5).

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 remembered that EUPHORIA is part of a distributed system; if the EUPHORIA module monopolizes the CPU, external modules on the same processor 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` [3] 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 [3]. 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.

Glossary

Configuration   The communication structure of an application. In this manual, the configuration refers to the constraint relationships among user interface graphics and/or the logical connections among distributed application modules.

Constraint   A relationship to be maintained among a set of graphics attributes and/or published variables. Usually, this relationship is satisfied by computing the value of an *output variable* as a function of a set *input variables*.

Data Boundary   In Playground, "data boundary" refers to a module's set of published variables (also known as the "presentation"). For a widget, "data boundary" refers its set of exposed attributes.

Distributed Application An application consisting of multiple concurrent processes, usually running on separate workstations, that communicate over a network. In Playground, an application consists of a collection of independent modules and a configuration of logical connections among the modules' published variables.

End-user   The user of an application. Here, end-users are people who are proficient in WYSIWYG applications, such as word processors and graphics editors, but are not necessarily experienced in textual programming.

EUPHORIA   End-User Production of grapHical interfaces fOR Interactive distributed Applications

I/O Abstraction   A connection-oriented model of interprocess communication in which independent modules interact with an abstract environment.

Logical Connection   The communication specification between two published variables. Connections can specify either bidirectional or unidirectional communication.

Module   A component of a distributed application. In Playground, a module is a process that has a set of published variables.

Multimedia   Integration of different media types such as text, interactive graphics, images, audio, and video within an application.

Published Variable   A data structure, such as a real number or an array, that is exposed from a module to the external environment. Whenever a published data structure is updated, communication to other published variables occurs implicitly according to the outgoing logical connections.

Acknowledgments

References

[1]   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, pp. 317-318, November 1995.

[2]   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, 21(9):735-746, September 1995.

[3]   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.

[4]   T. Paul McCartney. User Interface Applications of a Multi-way Constraint Solver. Washington University Department of Computer Science technical report WUCS-95-22, July 1995.

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