Mica Overview

This paper provides an introduction to Mica, a graphics library that can be used to manage all aspects of 2D graphics applications, including user-interfaces, diagrams, graphs and animations.

Mica is named after the finely-layered, flexible, partially transparent mineral that is found in nature.

This white paper refers to Mica Version 0.90 (Alpha)

About Mica

Mica is an object-oriented graphics framework specifically crafted to support the implementation and inter-mingling of graphing editors, drawing editors, and user interfaces. To this end Mica has extensive support for display lists, event handling, action dispatching, coordinate transforms, and connectivity.

Mica is not a desktop, though desktops can be implemented on top of Mica. Mica is not a user interface toolkit, though one is included with Mica. Mica is not just a object-oriented graphics toolkit, though there are primitive graphics objects within Mica: these graphics primitives have a wealth of functionality and there are many layers of functionality on top of them. Mica is not drawing editor, though one is included with it as a sample application. Mica is not an application framework, though Mica can be the graphics component of an application framework (see the forthcoming Cadabra white paper).

Mica is designed and written by programmers to support programmers. Whenever a part is moved or added or removed, an attribute is changed, the viewport modified, a button label changed, etc., Mica automatically updates any internal data, any lay-out, and the current view, if necessary. All parts derive from a richly featured base class in order that the programmer can easily add a bit more functionality to what may be historically considered a ‘simple’ part. Any part can be replaced by any

other part (for example a labeled icon in a node-arc graph can be replaced by a scrolled list or a plot or an embedded internal window). Similarly, a part can be assigned to another part as an attachment without having to alter the container-part hierarchy.

Acknowledgements

We want to thank Sun Microsystems for making such a fun programming language and for leading the battle for the rest of us in the portability wars.

We would also like to acknowledge those who have written and distributed graphics toolkit source code before us: we hope you enjoy cruising this code as much as we enjoyed cruising yours; those who have published papers, manuals and books about graphics toolkits: we all rely on you to propogate the art to the rest of us; and to those whose work and ideas are held hostage by their employers: we cast Mica into the winds as yet another blow against the empire.

This chapter presents an overview of Mica and how it works.

Design Goals

Mica is unashamedly designed for the programmer. As such the top priorities are:

1. Maximize the ease of use of the current features
2. Maximize the number (while maintaining the orthogonality) of features
3. Maximize the ease of adding features

Performance and memory size problems are tackled on a case-by-case basis if and when they arise.

Features

Written using Java (no native methods) and only a minimal amount of the AWT graphics API, Mica is extremely portable.

Many graphics objects are provided including shapes (line, rectangle, text,...), connections, widgets (push buttons, tables, tree lists, combo boxes,...), windows, dialogs and message boxes (using both native AWT frames and internal Mica frames), editors, choosers, pre-built menubars, toolbars, and graphics editors.

Subclassing from a single highly functional base class, all graphics objects therefore can be treated the same (reducing cognitive overhead), can be modified using their API or by using composition (preventing the need for of a lot of subclassing), and combined and used by other graphics objects without regard to their actual type.

The availability of the source code, for both the library and applications, makes it straight forward to mimic, copy-and-modify and debug.

A large number of behavioral objects, called event handlers, are provided which can be assigned to any graphics object. These are used, for example, by all Mica widgets to respond to the user's input. Each event handler has an event->functionality translation table which can be used to customize the precise behavior of any graphics object. The event handlers provided support, zoom, select, move, full-screen cursor, create connection, create text and much more. Special event handlers can be used to monitor and/or grab control of the event stream.

Events are Mica objects that contain useful information about the input event that generated them. All geometric information in an event is automatically transformed to the local space of each of the event handlers that examine the event. The event also contains the list of graphics objects that are lie underneath the point where the event occurred.

Actions (generated by graphics objects) are differentiated from events (generated the keyboard and mouse). Actions have four phases: request, cancel (when the request was vetoed), execute and commit.

World coordinates are used for all graphics objects for accuracy (using real numbers i.e. 'doubles') and display flexibility (magnification, birds-eye and fish-eye views, etc.). All transformations, which can be assigned to any container, are automatically used by Mica.

Any and all modifications made to any graphics object (whether to it's appearance, geometry, event handling or action handling) are automatically detected and accounted for by Mica. This includes but is not limited to updating layouts near the graphics object, updating the event and action masks of the graphics object and/or it's containers, and redrawing the graphics object.

Graphics objects are moved, resized, connected, reconnected, and animated in real-time so that the end-user does not get confused by 'disappearing graphics'.

Connections are first class graphics objects and extensive support for *having* connections is included in all graphics objects. Connections connect to graphics objects at common or any number of custom 'connection points'. Connections are automatically moved and updated by Mica. Connections are usually displayed as lines and can therefore have use of the dozen or so arrow heads and tails supplied.

Attachments are graphics objects that are assigned to other graphics objects but do not appear in the part-container hierarchy of a window. Attachments make it extremely easy to add ‘resizing handles’ to a selected graphics object or to add a textfield widget to a connection. Attachments can be assigned to a variety of positions with respect to their ‘host’ graphics object and this positioning is automatically maintained by Mica.

Any graphics object can be assigned a layout. Some layouts specify the positions of the parts inside a graphics object (i.e. a row layout), some specify the positions and connections of the parts inside a graphics object (i.e. a star graph layout), and some specify a constraint between a graphics object and another (i.e. x is to the left of y). All widgets use layouts to specify the positions of their constituent shapes.

Full support for end-user and programmatic specification of properties is provided. All text strings and icons displayed by Mica and it’s applications can be changed using the plain ASCII text files: *defaults.mica* and *properties.mica*. In addition, the default widget properties and any application-specific properties can also be set in these files. Every graphics object has all 60 or so of it’s attributes as properties in addition to any specific properties it may have. The translation tables of the event handlers assigned to a graphics object are also properties and in the future event handlers and widget prototype classes will also be able to be specified using properties (and property files).

Drag-and-Drop and Clipboard cut-copy-paste functionality is built-in to Mica. Any graphics object can be made a drag-and-drop source and/or target and actions for drag-over and drag-under effects are generated by the drag-and-drop manager.

Undo-redo-repeatable commands objects are used by the event handlers and a globally accessible ‘transaction manager’ collects these commands and manages them for programs written using Mica.

Extensive support for help is provided. Help can be assigned to any graphics object and can be a plain text string or a object that describes the text and the attributes of the text and background. The types of help are: toolhint (a smallish message), balloon (a larger toolhint with callout), statusbar (a message to be displayed in the status bar), dialog (a message to be displayed in a dialog box). The *helpviewer* class formats and displays a very simple formatted text file as a navigable help window.

Specialized renderers can be assigned to each graphics object. Default renderers are supplied with Mica. The types of renderers are: shadow, lineEnds, border, gradient, booleanState, before, after, background, and visibility.

Architecture - The Layered Approach

Mica is layered as follows, such that the lower layers know nothing of the layers above:

- Mica Editors
- Mica Part Assemblies
- Mica Widgets
- Mica Parts, Containers and Shapes
- Mica MiRenderer, MiCanvas
- java.AWT Graphics

The AWT Layer

Mica uses the drawing capability of the AWT Graphics class, the AWT Frame and Dialog classes for window handling, and AWT Canvas for drawing output and AWT Event handling. AWT Fonts and AWT Colors are using for rendering. Upon this is built a complete user-interface toolkit combined with a 2D vector graphics library and upon *this* are application-sized widgets with which one can easily create graphical applications.

The Mica-AWT Interface Layer

The AWT Graphics class is subclassed by MiRenderer which adds an API that supports drawing in world coordinates, device coordinate specialized renderers and the pushing and popping of override attributes. The AWT Canvas class is subclassed by MiCanvas and adds support for window locking and the event handling and animation thread.

The Mica Construction Layer

All graphics (shapes, widgets, choosers, editors, windows) in Mica are MiParts, which are arranged in groups using MiContainers. Shapes (like line, circle, rectangle, text,...) are used by more sophisticated parts to create their appearance.

The Mica Widgets Layer

This layer contains standard widgets which are built using shapes and other widgets.

The Mica Parts Layer

This layer contains large assemblies of widgets into tools like choosers (font, color, line width,...) and pre-built menus, toolbars and main windows.

The Mica Editors Layer

This layer contains pre-built editors for graphing, drawing and diagramming that can be used either as stand-alone windows or incorporated in other windows.

Event and Action Handling

Simply put: Mica manages an AWT Canvas in a AWT window, drawing Mica shapes and Mica widgets in the Canvas, watching for AWT Events generated by the user in the Canvas, converting them to Mica events, forwarding these events to the shapes and widgets, who generate Mica actions that larger assemblies of widgets do something intelligent with, just like the user intended them to do.

MiParts

Just about everything in Mica is a MiPart and Mica has been designed in order to provide many convenient ways to display, arrange, manipulate, inquire and interact with these MiParts.

MiParts are the basic geometric construction element in Mica. They have a name, are drawable, have attributes, receive and process events, generate actions, and many more capabilities. Mica has been intentionally designed to have all parts be very powerful, full-featured objects. This is in order to make programming with Mica easy and rewarding (when memory considerations become paramount, lightweight and very lightweight shapes can be used).

Through the use of containers and references, a part-container hierarchy can be constructed. This event and action propagations, drawing and other aspects of this hierarchy is automatically handled in Mica.Many traditional convolutions associated with programming GUIs are no longer required with Mica. Much of the tedious ‘housekeeping’ is handled by Mica itself, wherever possible. Examples of this are layout validation and invalidations, the enabling of actions and events, redrawing of shapes, etc.

MiPart Functionality Overview

This section lists the major areas of functionality of every MiPart and describes the basic idea and scope of each area.

Named Resources

MiParts have an unbounded array of named resources available for you to use and methods to set, get, remove and iterate through them.

Life and Death Management

MiParts have methods to *copy(), deepCopy(), deleteSelf(), removeSelf()* (from all containers), *replaceSelf(MiPart)* which are fully aware of any Attachments and Connections the part may have.

Deep Connections

MiParts have methods which support the iteration through all connections of the part and all of it's parts.

Drag and Drop Management

MiParts may be a source of and/or a target of a drag and drop operation. As such there are methods to indicate if such functionality is enabled (see MiAttributes), specify the drag and drop behavior, how the part will import and export data and what their valid data formats are.

Attributes

There are numerous methods to set and get individual attributes of a MiPart as well as it's assigned MiAttributes object.

Properties

Properties can be set and inquired and include all a MiPart's attributes and additional subclass-specific properties. In addition, a MiPropertyDescription can be obtained for each property that contains information about the type of the value of the property and list all valid values (if finite) and validate new values of the property.

Focus Management

Each part has the potential of having the current keyboard, mouse and/or enter-key focus. There are methods to request and inquire each kind of focus.

Select State, Sensitivity, and Visibility and Hidden State Management

There are methods to set and get the basic state of the MiPart.

Point Management

Methods to inquire, append, insert, and remove points are available for all MiParts. However, on parts that are not MiMultiPointShapes, the available points are the lower-left-hand and upper-left-hand corners and they can be inquired only.

Geometry Management

There are extensive methods to inquire and modify the geometry of every MiPart. This includes operations such as changing it's size and position. These methods are grouped into operations on the center, sides, height, width and bounds of the MiPart. In addition, basic operations such as translate, rotate and scale are available.

Pick Management

Pick management performs two functions: 1) indicating whether the MiPart intersects the given point and 2), returning a list of MiParts that intersect the given point.

Draw Management

MiParts have no draw methods that are available for your use; they are redrawn by Mica when their appearance or geometry changes. However there are methods to specify that the MiPart is to be drawn to and redrawn from a (double) buffer, to create an Image from an area of the MiPart, and to halt the current thread until the MiPart is redrawn (*waitUntilRedrawn*()). Note that a whole root window can be double buffered by using the specialized methods on their MiCanvas object.

Attachment Management

MiParts have methods to append, inquire and remove attached MiParts.

Container Content Management

All MiParts have methods to append, inquire and remove other MiParts. However these methods are only functional for MiContainers. Having MiPart implement these methods means a lot less of you having to explicitly test each MiPart to see if it is a MiContainer.

MiParts have methods that act on actual parts (*appendPart(MiPart))* and semantic parts: items (*appendItem(MiPart)*). Items are usually actual parts except in cases like MiLists (where an item is a row in the list), and like MiEditors with layers (where items are the shapes in the current layer).

Containers management

Methods are available to append, insert and inquire containers of the MiPart.

Bounds Management

Methods to set and get inner, outer and draw bounds and to set and get minimum and preferred sizes (which override those of any layout associated with the MiPart).

Invalid Area Management

Each MiPart has methods to invalidate areas within it's bounds, causing a subsequent redraw of the MiPart. This, however, rarely if ever needs to be used because Mica automatically invalidates areas that need it.

Other methods specify whether or not the MiPart is an *opaque rectangle* (the default is that it is not unless it is an instance of MiEditor, MiTable or MiMenu). If it is a opaque rectangle then nothing is drawn underneath the MiPart. The MiPart is assigned a draw manager that takes care of this. This is useful for both speed of execution and for aesthetics of appearance.

Layout Management

Provided are the methods to set and get the MiiLayout assigned to the MiPart and to invalidate and test the validity of any such layout. The ability to invalidate the MiPart's layout, however, rarely if ever needs to be used because Mica automatically invalidates layouts that need it.

Connection Management

MiConnections can be appended, inserted, removed and inquired. Convenience methods are available to get all of a MiParts parents and children and to return whether of not the MiPart is connected to another, given, MiPart.

Connection Point Management

A MiConnectionPointManager can be assigned to the MiPart (See chapter on Connections).

Event Handling

Any number of event handlers can be assigned to any MiPart and MiParts have methods to append, insert and remove and enable/disable event handlers.

If a event handler is assigned to the MiPart and is not position dependent then it is automatically registered with the MiPart's window (if and when it has a containing window) as a global event handler (i.e. a hot key/accelerator event handler. Similarly it will be automatically removed from the window if the event handler is removed from the MiPart or if the MiPart is removed from the window).

There are also methods that inquire what events the MiPart (i.e. all of it's event handlers) is interested in.

Action Handling

A large number of methods are provided to append, insert, and remove action handlers (actually the MiiActions that are to be dispatched to the MiiActionHandler are what are registered; see the chapter on Actions). A number of methods are also available to register callbacks, which are sometimes more convenient to code that action handlers and which simply send a text String to a MiiCommandHandler object.

Action Generation

A number of actions are generated directly by the MiPart class. Some of these are generated only when there is a action handler registered that is interested in the action. These will be marked with a *. The others will be generated and iterate through each action handler assigned to the MiPart looking for an interested handler. These others will then check a special composite handler that represents the action handlers of all of the MiPart's containers and their containers, etc. If this composite handler is interested, then the action is forwarded up the part-container hierarchy. The actions generated by the MiPart are:

- Mi_COPY_ACTION
- Mi_REPLACE_ACTION
- Mi_DELETE_ACTION
- Mi_GOT_KEYBOARD_FOCUS_ACTION
- Mi_LOST_KEYBOARD_FOCUS_ACTION
- Mi_GOT_ENTER_KEY_FOCUS_ACTION
- Mi_LOST_ENTER_KEY_FOCUS_ACTION
- Mi_GOT_MOUSE_FOCUS_ACTION
- Mi_LOST_MOUSE_FOCUS_ACTION

- Mi_SELECTED_ACTION
- Mi_DESELECTED_ACTION
- Mi_HIDDEN_ACTION
- Mi_UNHIDDEN_ACTION
- Mi_PART_VISIBLE_ACTION
- Mi_PART_INVISIBLE_ACTION
- Mi_INVISIBLE_ACTION
- Mi_VISIBLE_ACTION
- Mi_DRAW_ACTION*
- Mi_SIZE_CHANGE_ACTION*
- Mi_POSITION_CHANGE_ACTION*
- Mi_GEOMETRY_CHANGE_ACTION*
- Mi_APPEARANCE_CHANGE_ACTION*

Manipulator Management

These few methods support two kinds of manipulators: part manipulators and layout manipulators. For each of these manipulators there is a method to create the manipulator (for the MiPart or it's layout) and a method to get the manipulator that has already been assigned to the MiPart (or it's layout) if any.

Special Containers Management

MiParts have 3 methods that return important containers of the MiPart. These methods are:

MiWindow	*getRootWindow*()
MiEditor	*getContainingEditor*()
MiWindow	*getContainingWindow*()

Debug Management

There are a number of methods that are dedicated to helping track what is happening to the MiPart. For example there is a getID() method that will a unique integer identifying the MiPart.

The MiPart Class Hierarchy

MiPart

- MiArc
- MiCircle
- MiEllipse
- MiEllipticalArc
- MiImage
- MiLiteShapesContainer
- MiMultiPointShape
  - MiLine
  - MiPolyline
  - MiPolyPoint
  - MiPolygon
  - MiTriangle
- MiRectangle
- MiRoundRectangle
- MiText
- MiVeryLightweightShape
- MiContainer
  - MiLayout
    - MiManipulatableLayout
      - MiRowColBaseLayout
        - MiColumnLayout
        - MiRowLayout
      - MiGridLayout

      - Mi2DMeshGraphLayout
      - MiCrossBarGraphLayout
      - MiLineGraphLayout
      - MiOmegaGraphLayout
      - MiOutlineGraphLayout
      - MiRingGraphLayout
      - MiStarGraphLayout
      - MiTreeGraphLayout

MiUndirGraphLayout
MiPolyLayout
MiPolyConstraint

MiEditor
MiWindow
MiNativeWindow
MiNativeDialog
MiNativeMessageDialog
MiInternalWindow
MiDialog
MiMessageDialog

MiVisibleContainer
MiWidget
MiAdjuster
MiSlider
MiGauge
MiScrollBar
MiAttributeOptionMenu
MiBorderLookOptionMenu
MiColorOptionMenu
MiFontOptionMenu
MiFontPointSizeOptionMenu
MiLineEndsOptionMenu
MiLineWidthOptionMenu
MiBox
MiLabel
MiButton
MiCheckBox
MiCircleToggleButton
MiMenuLauncherButton
MiOptionMenu
MiPushButton
MiSpinButton
MiToggleButton

- MiMenuItem
- MiColorChooser
- MiComboBox
- MiExpandoBox
- MiFontChooser
- MiFileChooser
- MiLabeledWidget
- MiMenu
- MiMenuBar
  - MiEditorMenuBar
- MiOkCancelHelpButtons
- MiPieChart
- MiPlayerPanel
- MiPropertyPanel
  - MiBasicPropertyPanel
  - MiComboPlusPropertyPanel
  - MiListPlusPropertyPanel
  - MiTablePropertyPanel
- MiRadioBox
- MiScrolledBox
- MiStandardMenu
- MiStatusBar
  - MiEditorStatusBar
    - MiBasicStatusField
    - MiCurrentTimeStatusField
    - MiMagnificationStatusField
    - MiMouseXYPositionStatusField
    - MiCurrentTimeStatusField
    - MiMagnificationStatusField
    - MiMouseXYPositionStatusField
- MiTabbedFolder
- MiTable
  - MiList
  - MiTreeList
- MiTextField
- MiToolBar

MiEditorToolBar
MiWindowBorder

LightweightShape Class Hierarchy

MiLightweightShape
- MiArcLite
- MiCircleLite
- MiEllipseLite
- MiImageLite
- MiLineLite
- MiPointLite
- MiPolyLineLite
- MiPolyPointLite
- MiPolygonLite
- MiRectLite
- MiTextLite

Pre-Built Menu Class Hierarchy

- MiiContextMenu
  - MiEditorMenu
    - MiConnectMenu
    - MiEditMenu
    - MiFileMenu
    - MiFormatMenu
    - MiGraphMenu
    - MiHelpMenu
    - MiLayoutMenu
    - MiShapeMenu
    - MiToolsMenu
    - MiViewMenu
      - MiEditorBackgroundMenu

The Command Handler Class Hierarchy

MiiCommandHandler
    MiiTargetableCommandHandler
        MiCommandHandler
            MiLayoutPartsCommand
            MiPanAndZoomCommand
            MiConnectMenuCommands
            MiReorderPartsCommand
            MiDeletePartsCommand
            MiEditMenuCommands
            MiEditorBackgroundMenuCommands
            MiFileMenuCommands
            MiFormatMenuCommands
            MiGraphMenuCommands
            MiGraphPartsCommand
            MiGroupPartsCommand
            MiHelpMenuCommands
            MiIExecuteCommand
            MiIconifyPartsCommand
            MiLayoutMenuCommands
            MiReplacePartsCommand
            MiSelectPartsCommand
            MiShapeMenuCommands
            MiShapePopupMenuCommands
            MiToolsMenuCommands
            MiTransactionCommandAdapter
            MiTranslatePartsCommand
            MiViewMenuCommands

The Event Handler Class Hierarchy

MiiEventHandler
    MiEventHandler

MiIClickAndDrop
MiICreateConnection
MiICreateMultiPointObject
MiICreateObject
MiICreateText
MiIDeleteObjectUnderMouse
MiIDeleteSelectedObjects
MiIDeselectAll
MiIDisplayContextCursor
MiIDisplayContextMenu
MiIDisplayHelpDialog
MiIDisplayToolHints
MiIDragAndCopyWithMouse
MiIDragBackgroundPan
MiIDragObjectUnderMouse
MiIDragSelectedObjects
MiIDragger
MiIExecuteActionHandler
MiIExecuteCommand
MiIFlowEditorEventHandler
MiIFullScreenCursor
MiIJumpPan
MiIMouseEnterAndExit
MiIMouseFocus
MiINormalizedPan
MiIOnePtPan
MiIPan
MiIPartInspector
MiPlayEventSound
MiIPopup
MiISetDebugTraceModes
MiIPrintGraphicsStructures
MiIPrintPostScript
MiIReCalcLayouts
MiIRedraw
MiIRubberbandBounds

MiIRubberbandPoint
MiISelectArea
MiISelectObjectUnderMouse
MiIZoomArea
MiIZoomAroundMouse