Reference Manual
For
The MRE Graphics Interface

By Mark L. James and Doug Lyon
Last Update: 5/28/86

Jet Propulsion Laboratory
California Institute of Technology
About This Manual

This manual describes the MRE-Graphics system. You can use it now to learn the basic operation of the MRE-Graphics system and use it later as a reference manual. This manual tells you how to:

- Load the MRE-Graphics system.
- Initialize the MRE-Graphics system.
- Create MRE-Graphics screens.
- Run MRE-Graphics demonstrations.
- Use the MRE-Graphics package in user applications.
Introduction

The MRE (Multiple Reasoning Engine) graphics application package is intended for programmers interested in drawing interactive graphics with an emphasis on the drawing and placement of graphs. The reasoning engine programmer is likely to find the MRE-Graphics package useful for displaying inference chains to aid in debugging production rules.
Contents

• Loading the MRE-Graphics-system
• Package Naming Convention
• Initializing The MRE-Graphics System
• Trying Some Example Functions
• Executive MRE-Graphics Cookbook
• Windows
• Important User Interface Functions
• Important User Interface Messages To Graphic Space Instances
• More Functions
• The Calculation-Mixin Flavor
• The Connection Flavor
• The Drawing-Mixin Flavor
• The Dynamic-Placement-Mixin Flavor
• The Graphics-Space Flavor
• The Intersection-Grid-Mixin Flavor
• The Kinetic-Mixin Flavor
• The Node-Mixin Flavor
• The String Flavor
• The Transformable-Graphics-Object-Mixin Flavor
• Creating Your Own Node Shapes
• Glossary
Loading the MRE-Graphics system

To load the MRE-Graphics system the user types:
(load "sun:/yon>graphics>makesys.lisp")
(make-system 'graphics :silent :noconfirm :nowarn)

Loading the Color system

Prerequisite: The Color System

To load the color system type:
(make-system 'color :nowarn :noconfirm)

Package Naming Convention

The package name of the MRE-Graphics system is MRE-Graphics:. The MRE-Graphics system is nicknamed MRE-G: and the shorter nickname will appear in front of all non-exported symbols.

Initializing The MRE-Graphics System

Prerequisite: Loading the MRE-Graphics-system.

Initializing the MRE-Graphics-system will mean different things to different people. In order to bring up a black and white graphics-window which contains a graphics-pane and a type-in-pane on the standard lisp machine console type:
(MRE-G:graphics)
If you want the color system (and it has been loaded) type:
(MRE-G:graphics t)
Instead, This will perform a process-run-function and return immediately, the user will notice increased run bar activity. When the function is finished initializing the black and white graphics window the user will see a bordered-constraint-frame window with two panes.
The MRE-G:graphics function is called only once at the beginning of a user’s session and sets up the default window. It allows the user to type:
<select>-g
To return to the graphics-window after selecting some other window. The first time (MRE-G:graphics) is typed the default graphics window is selected for the user automatically. Now the user is ready to try some example functions.

Making Your Own Function For Initializing The Graphics System

The experienced user may wish to run the graphics system without going through the standard procedures as described above. Reasons for this may include the setting up of a custom graphics pane, multiple graphics spaces, special mouse sensitivity, etc. Typically what initializing the graphics system will do is set a global variable called MRE-G:Graphics-Frame before default functions like
MRE-G:Create-My-Window and the examples will work. If the user does not want to set the MRE-G:Graphics-Frame then the default for the :Map-To-Window on the MRE-G:Create-Graphics-Space function will not work and the user will have to provide a :Map-To-Window.
Trying Some Example Functions

Prerequisite: Initializing The MRE-Graphics-system.

A type in pane labeled New Age Type In Window Concepts appears at the bordered constraint frame bottom. To run the following example type into the type in pane:

(MRE-G:graph 2)

This will invoke the following code:

```
(defun graph (n &key (color? nil)
  (connections :random)
  (node-shape 'circle))

;;; 'rectangle is also possible.
(let ( my-window g rp cp )
  (setq levels n)
  (setq my-window (create-my-window color?)))
(if g-space (send g-space :kill-mouse-process))
(setq g-space (create-graphics-space 'g-space :map-to-window my-window))
(setq g-space :clear)
(setq g-space :focus-on-active-object :switch :off)
(setq cp (make-object-desc node-shape))
(setq mother (send g-space :create-object cp :x-coord 500 :y-coord 100 :string "mother"))
(btree g-space cp mother 1 connections)
(send g-space :default-menu)
(send g-space :start-mouse-process)))
```

Explaining The Example

(MRE-G:graph n &key (color? nil)
  (connections :random)
  (node-shape 'circle))

- $n$ is an integer used to indicate the number of levels in the graph to be generated.

- color?
  When non-nil color? will cause the output of the MRE-Graphics system to appear on the Color System. Prerequisite: Initializing The Color System.

- connections may have the value
  
  - :random
    This will cause nodes to be connected randomly with the restrictions that every node is connected at least once, is never connected to the same node twice and is never connected to itself.
° :complete
  If this is used then the graph of n nodes is drawn such that the nodes are completely connected.

• :node-shape <Type>
  Type may be 'MRE-G:Circle or 'MRE-G:Rectangle.

• My-window is the graphics-space window created by default function MRE-G:create-my-window.

• Cp is a description of a circle.

• G-space is a global variable. See :kill-mouse-process method of graphics-space flavor for more information.

The Tree Example

To run the tree example type:
(MRE-G:tree 2)
This will draw a widely spaced tree. To obtain a more dense tree mouse-right on the mother and select the "Squish Down" option from the menu.

The Textured-Tree Example

To create a tree identical with the tree in the last section but with textured connections type:
(MRE-G:textured-tree 2)
The user is invited to examine the code of these last two examples to notice that the only difference is the use of the :texture and :width keys when the objects are connected. See the :Connect-Objects method in the Important User Interface Messages To Graphics Space Instances section of this manual.

The Multi-Growth-Tree Example

Here is a tree which uses placement by numerical methods. A node is moved through a user specified angle at an ever increasing radius until a free space is found. This is very CPU bound but can result in good placement density. To run this example the user types:
(MRE-G:Multi-Growth-Tree 2)
See the Important User Interface Messages To Graphics Space Instances section for more information about the level option.
Executive MRE-Graphics Cookbook

Prerequisite: Initializing The MRE-Graphics-system.

Described here is a step by step description of how to create and connect two objects using the MRE-Graphics package.

What To Do

1. \((\text{Setq } W (\text{MRE-G:Create-My-Window}))\)
   \[\text{Use } (\text{Setq } W (\text{MRE-G:Create-My-Window } t))\]
   \[\text{If you want and have loaded color.}\]

2. \((\text{Setq } G (\text{MRE-G:Create-Graphics-Space } '\text{Gspa :Map-To-Window } W))\)

3. \((\text{Setq } Od (\text{MRE-G:Make-Object-Desc } '\text{Mre-G:Rectangle}))\)

4. \((\text{Setq } O1 (\text{Send MRE-G:G :create-object od :string } "\text{Node #1}"))\)

5. \((\text{Send } O1 :\text{Move } 500 400)\)

6. \((\text{Setq } O2 (\text{Send MRE-G:G :Create-Object Od :String } "\text{Node #2}"))\)

7. \((\text{Send } G :\text{Connect-Objects } O1 O2)\)

What It Means To Do It

1. This sets \(W\) to an instance of a graphics pane. This is the top pane in the bordered-constraint-pane labeled "New Age Graphics Space Concepts".

2. This sets \(G\) to an instance of a graphics space. The instance name is arbitrarily named \(\text{gspa}\).

3. This sets \(Od\) to an instance of an object description of \(\text{MRE-G:Rectangle}\) type.

4. This sets \(O1\) to an instance of a graphics space object labeled "Node #1" and draws this object on the window called \(W\).

Recovering From Bad Errors

Prerequisite: Initializing The MRE-Graphics-system.

Occasionally the user will find it neccessary to reset the graphics-window and regenerate the bordered constraint frame. To do this the user types:

\[
\text{MRE-G:reset \&optional } \quad \text{(color? nil)} \quad \text{(graphics-pane-name } "\text{graphics pane}" ) \\
\text{Function}
\]

create a new bordered constraint frame black and white window or a new color window and bind this window instance to the global variable called \(\text{Graphics-Frame}\).
For more about window see the following section.
Windows

The window flavor descriptions are intended to support the example programs and serve as a basis for the user's own custom window flavors. It is the stated intention of the design of the MRE-Graphics package not to assume the window needs of the user. However, the user is cautioned to mixin the MRE-G:Users-Graphics-Window-Mixin because of certain assumed windowing capabilities.

- **bordered-window** has tv:bordered-constraint-frame and tv:window mixed-in. There are usually at least 2 panes in a bordered window, a graphics pane and a type in pane.


- **users-graphics-window-mixin** has tv:basic-mouse-sensitive-items, tv:stream-mixin and tv:window mixed in.

Important User Interface Functions

The following functions are vital in most user applications and have been designated the official method for user interface with the MRE-Graphics system.

- **MRE-G:Create-Graphics-Space Name &Key**
  
  \[\text{Map-to-window (create-my-window)}\]
  
  \[\text{default-font fonts:cpffont}\]
  
  Function

  Creates and returns a **Graphic-Space** instance.

- **Name** is the name of the graphics space.

- The following keyword options are possible:
  
  - **:Map-To-Window <Window-Instance>**
    
    Specifies that the graphics space is to be displayed on the window **Window-Instance**.

  - **:default-font <Font>**
    
    Here the font may be any font loaded into the system.

- **MRE-G:Make-Object-Desc Type &key**

  \[\text{(height 20)}\]
  
  \[\text{(width 20)}\]
  
  \[\text{(string nil)}\]
  
  \[\text{(font fonts:cpffont)}\]
  
  Function

  Returns a datatype that is an instantiated version of a generic object, i.e. its length, width, etc. have been defined.

  **Type** can be from any of the following:

  - **MRE-G:Rectangle**
  
  - **MRE-G:Circle**
  
  - **MRE-G:Fabricated-Object**

  Look under the **Flavors** section to find out the methods and instance variables for each one. The object here is to make a template object description for the graphics space instance to copy. This allows the user to define the defaults since all settable variables will be copied from the objects description instance.
Important User Interface Messages To Graphic Space Instances

The following messages are vital in most user applications and have been designated the official method for user interface with the MRE-Graphics system. For more information see the Graphics Flavor section.

- **:Kill-Mouse-Process** of MRE-G:Graphics-Space
  
  Method
  
  It is recommended that an old Graphics-space instance be sent a :kill-mouse-process message before a new graphics-space is created. Every graphics-space instance has a window which it sends draw commands to. This window is an instance of a graphics-window-flavor and is discussed in the Windows section of the manual.

- **:Start-Mouse-Process** of MRE-G:Graphics-Space
  
  Method
  
  Typically a user starts a mouse process just before application program completion. This mouse process sends an :any-tyi message to the map-to-window. If another graphics-space instance is made with the same map-to-window and if a mouse-process is started for this graphics-space the user will generate an error. It is the users responsibility to kill a mouse process associated with an old graphics-space if a new graphics-space instance is created with the same map-to-window.

- **:Create-Object <Desc> &Key
  
  (:X-Coord 500)
  (:Y-Coord 500)
  (:String "") of MRE-G:Graphics-space
  
  Method
  
  This method creates an instance of the graphic object described by Desc and draws it in the graphic space. Create-Object returns an instance of the description returned by Make-Object-Desc the format for a desc is an instance so this method returns a specially instantiated copy of the instance desc.

  The following keys are legitimate:

  - **:X-Coord <n>**
    
    The object is to be placed at the X coordinate of n.

  - **:Y-Coord <n>**
    
    The object is to be placed at the Y coordinate of n. Generally the x and y coordinates of a object define the objects center. For user defined objects and for fabricated objects this may not necessarily be true.

  - **:String <string>**
    
    The string is to serve as a label for the object with will have meaning to the user.

- **:Connect-Objects <Obj1> <Obj2> &key
  
  (:width 5)
  (:texture nil)
  (:entry-points :discrete)
Connects Obj1 to Obj2 by a user specified line segment. The result of this message is an instance representing the line that was drawn. Connect-Objects automatically adds Obj1 to the list of parents contained in Obj2 and adds Obj2 to the list of children in Obj1 if it is appropriate to do so. Before a parent child relation is formed a test is given to see if the proposed child will be "allowed" in the family. The childs' blood line is examined and if it is found that the child is already an ancestor of the parent (however remote) the child is barred from admission. This is done to protect the program internals which perform operations on trees via recursive mechanisms. Go ahead and add the children by hand but do so at your own risk. A special data structure exists called the nodes-to-be-placed-data-structure, this allows for incestual family relationships but uses iterative placement techniques which require a substantial increase in CPU usage.

° Obj1 and Obj2 are instances of the two objects to connect. The instances are those returned from :Create-Object. Below is a set of keys that can affect the operation of this message:

° :width <N>
  This is the width of a connection. This will not be used if the texture is left unspecified.

° :texture <Texture>
  Texture may be any texture such as tv:hes-gray. There is a small, linear increase in CPU usage when textures are used. The texture of a connection will be a gray array which is bit-blited onto a scratch screen and added with a rectangle which is in the shape of the connection. This is then blited onto the graphics pane.

° :Entry-points <Flag>
  Flag may be:

  • :Discrete
    Each connection will enter a node at a point on the node perimeter which is North, South, East or West of node center. The program automatically uses the shortest path to decide where to enter the node.

  • :Continuous
    the node entry point will "float" around the perimeter of the node attempting to shorten the distance between the node entry point and the center of a ring-placement topology. Using the :Continuous option for non-circular nodes or non-ring topologies may yield incorrect results.

° :Level <Flag>
  Flag may be any of the following:
• **Non-NIL**
  
  *Obj1* and *Obj2* are placed on the same level.

• **NIL**
  
  Then the level constraint is relaxed and space is searched for in a manner more likely to yield compact results. See the *max-number-of-trials-for-swing-placement* instance variable in the *Graphics-Space Flavor* section of this manual.

• **:Place-Now? <Flag>**
  
  This is used to forstall placement. It can work indefinitely. The intended use for this feature is to allow many node to be connected and semantically related with respect to their placement. Thus the user may build a graph and define the relationships between node position without having to invoke placement. This is also useful for connecting nodes in graphs with already acceptable placement (as in the introduction of siblings in an untangled tree).

• **:Placement <Flag>**
  
  *Flag* may be:

  ° **:Left**
    
    *Obj2* is placed to the left of *Obj1*.

  ° **:Right**
    
    *Obj2* is placed to the right of *Obj1*.

  ° **:Up**
    
    *Obj2* is placed to the up of *Obj1*.

  ° **:Down**
    
    *Obj2* is placed below *Obj1*. The above 4 flags automatically update the direction-data-structure and avoids insestual checkups when connecting a family. The penalty is greater CPU usage during placement.

  ° **:place-children-up**

  ° **:place-children-down**

  ° **:place-children-left**

  ° **:place-children-right**
  
  These 4 messages check for insestual relationships before doing tree placement. They are much faster because of the geometrical algorithm used in placement. It is less general however because the children in a graph may not be directly connected to the siblings unless the placement is *:inhibit* for the offending connections. Since these are immediate mode commands the user may not mask there usage with the *:Place-Now?* flag. These commands do placement by finding the width of the greatest grandchildren from *Obj2* and create a tree which can easily accomidate all the children. Better packing of objects onto the screen can be
obtained after these commands are used if the user send the appropriate squish message.

° :inhibit
No placement is recorded in the placement data structure but a connection is drawn and placement will occur if :Place-Now? is Non-Nil.

° :ring-topology
Places the nodes in a ring. The user is advised to make :entry-points :continuous when using this option.

° :Connecting-Angle <Type>
This option specifies at what angle this connecting line is to be drawn. Type is selected from below:

  • :Diagonal
    The connecting line can be a diagonal between the two objects. This is the default.

  • :Spline
    The connecting line can be a spline. The spline is constrained by 1st derivative continuity with a normal to the node surface and its' endpoints touch the node entry points so that a bezier curve instead of a spline. Small linear increases in CPU usage will be noticed.
More Functions

- **MRE-G:Angle-Between-Objects O1 O2**
  Returns a degree angle between objects.

- **MRE-G:Atand dx dy**
  Returns a degree result.

- **MRE-G:Any-True? List**
  If any of the List is non-nil the return is T.

- **MRE-G:Broadcast ObjectList Message (MessageParams) &Key**
  (return :deep)
  (concurrent? nil)
  (priority -10)

This send Message with the optional message parameters to every object in ObjectList. Use the concurrent option with caution as the length of the objects will determine how many processes will be started. This can create a hazard. The following keys are permitted:

- **:Return**
  whose value may be:

  - **:None**
    No returns are collected.

  - **:Deep**
    Returns are placed in a list of lists.

    - **:Flat**
      Returns are placed in a flat list suitable for rebroadcast.

- **:Concurrent?**
  whose values may be:

  - **NIL**
    The process is run sequentially, each send waits for a return in sequence.

  - **Non-NIL**
    The process is run concurrently, each send returns immediately after starting a process.

- **:Priority**
  An integer, not to high, to be used as process priority.

- **MRE-G:Create-My-Window &optional (color? nil)**

  Color? may take on the following values:

  - **NIL**
    Used for a monocrome window.
- Non-nil
  Prerequisite: Initializing The Color System.
  This will cause the MRE-Graphics output to appear on the Color System.

- MRE-G:Copy Instance
  Function
  This takes an instance and returns a new instance with the same values in
  the instance variables. Beware that the which operations message is not updated
  every time a new instance is made. This can cause confusing errors the only
  known cure for which is a cold boot. Basic-copyable-object must be mixed-in.

- MRE-G:Get-Mouse G-Space
  Function
  Given the graphics space instance G-Space Get-Mouse will get the mouse in an
  infinite loop.

- MRE-G:Hardcopy-files Direc
  Function
  Direc is a path name with optional wild cards and optional .newest embedded.
  This reverse spools files out to the laser printer.

- MRE-G:ld direc &key (direction :fowards) (return-list? nil)
  Function
  LD (List Directory).
  Direc is a path name with optional wild cards and optional .newest embedded.
  Direction may be any of the following:
  - :Fowards
    Lists the files in lexicographic order.
  - :Backwards
    Lists the files in reverse lexicographic order.

- Return-List?
  May be any of the following:
  - NIL
    Then LD will not return a list of files and output instead will appear on
    terminal-io.
  - Non-NIL
    Then LD will return a list of files.

- MRE-G:List-Com Object &optional String (return-list? nil) (exception-string nil)
  Function
  Does a :which-operations to Object and forms a list-of-operations.
  If String is specified a new list is formed called filtered-list. The filtered-list is
  formed by matching String with the elements in the list-of-operations. If
  exception-string is present then the elements in the filtered-list with matching
  substrings are removed. Finally if Return-list? is nil the filtered-list is printed,
  otherwise the filtered-list is returned.

- MRE-G:List-Methods
  Function
  Returns all methods from a :which-operations which do not have the set-
  prefix.
• **MRE-G: Radians-To-Degrees Theta-In-Radians**  
  Takes a radial angle and returns degrees.

• **MRE-G: Retrieve-String &Key**  
  (Prompting-String "Please Enter Your String")  
  (Default-String "Type In Some Jazz Here")  
  This uses a pop-up window to ask the user to enter a string.

• **MRE-G: Sum A-List-Of-Numbers**  
  This takes a list of numbers and returns their sum.

• **MRE-G: take-first n some-list**  
  Returns the first n elements from the list in the second argument.
The Calculation-Mixin Flavor

A primary flavor with several dependents. The calculation-mixin is mixed in to give an object some common capabilities. It is used as a special library of software tools.

Messages

The following are messages to the calculation-mixin:

- **:dilate u x1 y1** of MRE-G::*flavor*  
  Returns a number which is u percent between x1 and y1.

- **:find-distance-between-two-points x1 y1 x2 y2** of MRE-G::*flavor*  
  Returns the euclid distance between two points.

- **:print-array** of MRE-G::*flavor*  
  Uses the instance variables px and py which are points in a graphics object.
The Connection Flavor

"Connection" is a flavor which is usually used internally by the MRE graphics package. This documentation exists largely for maintainability of the package and to satisfy the curious.

The following instance variables are usually set automatically by the :connect-objects message which is handled by the graphics-space flavor.

- (object1 nil)
- (object2 nil)
  These are the objects to be connected by the connection instance.
- (level nil)
  Now believed to be obsolete, level used to be used by an auto-placement algorithm and would attempt to make object1 and object2 level with each other.
- (points-to-object1? nil)
  Indicates if arrow1 is to point to Object1.
- (points-to-object2? nil)
  Indicates if arrow2 is to point to Object2. If these flags are non-nil then the arrows are reoriented and redrawn when the connection is redrawn.
- (draw-straight-alu tv:alu-ior)
  This is the alu function used for drawing straight lines. A straight line is a line which is always slope 0 or ∞. It is used to make connections between objects such that there length is always a Manhatten distance. Tv:alu-ior always sets a pixel on despite its previous state.
- (erase-straight-alu tv:alu-andca)
  This always sets a pixel off regardless of its' previous state. Using these destructive alu functions has been found to be a necessary evil. Necessary because straight line connections often write over each other. Evil because dragging a connection which is destructively updating the screen requires an entire screen refresh, this can be computationally expensive and annoying.
- (x1 nil)
- (y1 nil)
  These are the coordinates of Object1’s center.
- (entry-point-x1 nil)
- (entry-point-y1 nil)
  This is the point at which the connection instance will intersect Object1.
  (entry-point-x2 nil) (entry-point-y2 nil) This is the point at which the connection instance will intersect Object2.
• (arrow1 (make-instance 'arrow))

• (arrow2 (make-instance 'arrow))
  These are arrows which can be drawn at run time. Arrow1 is designed to point
towards Object1. Arrow2 is designed to points towards Object2.

• (connection-type nil)
  This can be :diagonal :straight or :spline. :diagonal goes from entry point to
  entry point. :straight is :diagonal at right angles. :spline is :diagonal with 1st
derivative continuity.

• (spline-array-x nil)

• (spline-array-y nil)
  Points in the spline curve.

• (number-of-points-in-spline nil)
  Number of points in the spline curve.

• (angle-between-object1-and-2 nil)
  An internal variable in degrees.

• (relative-orientation nil)
  With respect to Object1 and Object2. This takes on the values:
    ° :left-to-right ==> Object1 is left of Object2.
    ° :right-to-left ==> Object1 is to the right of Object2.
    ° :top-to-bottom ==> Object1 is above Object2.
    ° :bottom-to-top ==> Object1 is below Object2.

• (growth nil)
  Presently an obsolete means of specifying placement.

• (window terminal-io)
  Generally this will be set to be the Map-to-window in the Graphics-space.

• (label (make-instance 'string))
  This is the label on the connection.

• (label-clipping? nil)
  When this is non-nil the label will be shortened to fit on the connection. **Not
  yet implemented**

The Connection Flavor Mixins

• connection-label-mixin

• drawing-mixin
• calculation-mixin
• si:property-list-mixin
Messages to Connections

- **:bottom-to-top-connect top-object bottom-object** of MRE-G:Connection Method
  These make a simple connection between Object1 and Object2. These routines are usually called by a program after the relative orientation of the objects is calculated.

- **:bottom-to-top-spline-connect Top-object Bottom-object** of MRE-G:Connection Method
  The above methods are called by a routine which has calculated the relative orientation of the objects.

- **:calculate-bottom-to-top-entry-point** of MRE-G:Connection Method
  Assumes Object1 is below Object2 and calculates both entry points.

- **:calculate-entry-points** of MRE-G:Connection Method
  Figures out the entry-points for both objects. The entry points are another name for the end points of the connection.

- **:calculate-left-to-right-entry-point** of MRE-G:Connection Method
  Assumes Object1 is left of Object2 and calculates both entry points.

- **:calculate-relative-orientation** of MRE-G:Connection Method
  Properly set the relative orientation instance variable depending on the relative position of Object1 and Object2.

- **:calculate-right-to-left-entry-point** of MRE-G:Connection Method
  Assumes Object1 is right of Object2 and calculates both entry points.

- **:calculate-top-to-bottom-entry-point** of MRE-G:Connection Method
  Assumes Object1 is above Object2 and calculates both entry points.

- **:delete** of MRE-G:Connection Method
  Sends objects 1 and 2 the :delete-connection message and proceeds to erase itself from the screen. Handles for connections are only stored in the objects being connected.

- **:diagonal Object1 Object2** of MRE-G:Connection Method
  A simple connection is made between the objects.

- **:draw** of MRE-G:Connection Method
  Clips label and sends connection instance the appropriate connection-type command. Please note that the connection-type is an instance variable which has the same content as an appropriate message name used for drawing the connection.

- **:draw-arrows** &optional (which-arrows :both) of MRE-G:Connection Method
  Which-arrows can have the value:
  - :both - draws both arrows
- `:arrow1` - draws only `arrow1`
- `:arrow2` - draws only `arrow2`

- **:draw-straight x1 y1 x2 y2 alu** of MRE-G:Connection
  A lower level draw function which draws the straight connection type after calculating the relative orientation.

- **:draw-straight-old x1 y1 x2 y2 alu** of MRE-G:Connection
  Draw based on relative orientation. Does not calculate the relative orientation first. This is because the objects might be in motion (this is what differs an erase from a draw). You see if the object is moving and we need to update the connection image we want to draw the old connection with an alu function which will draw over the old connection precisely. This will hopefully erase the old connection. When we redraw we do it with the draw-straight method because this will recalculate the orientation....

- **:draw-straight-specific x1 y1 x2 y2 alu draw-key** of MRE-G:Connection
  Recall that a straight line is what is come to be known as a "T-Bar connection". The nature of the T-bar connection is such that we can have a change in one dimension followed by a change in another dimension. This is encoded in the draw-key. Draw-key may have the following parameters :dx:dy:dx - change of x first then change in y then change in x. Or... :dy:dx :erase of MRE-G:Connection

- **:horizontal-connect** of MRE-G:Connection
  Connects Object1's left or right entry point to Object2's left or right entry point.

- **:horizontal-spline-connect Object1 Object2** of MRE-G:Connection

- **:left-to-right-connect left-object right-object** of MRE-G:Connection

- **:left-to-right-spline-connect Left-object Right-object** of MRE-G:Connection

- **:length** of MRE-G:Connection
  Returns the distance from Object1 to Object2 in pixels.

- **:New-String <String> &Key (Font fonts:cptfont)** of MRE-G:Graphics-Space
  Causes the indicated object to be labeled. String is used as the string to label the connection.

  - `:Font`<Font>
    Specifies the font of the label, the default is fonts: cptfont.

- **:orient-arrow1** of MRE-G:Connection
  **:orient-arrow2** of MRE-G:Connection
  Rotates the arrows to the correct position.

- **:prep-arrows** of MRE-G:Connection
  Causes window inheritance from the connection. Orient the arrows and
temporarily draws them.

- \texttt{record-connection-profile} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  Stashes the positions of the objects.

- \texttt{redraw} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  Redraws self, labels and arrows.

- \texttt{redraw-arrows} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  Redraws the arrows if the Points-to-object booleans are non-nil.

- \texttt{right-to-left-connect left-object right-object} of MRE-G:Connection \hspace{1cm} \textbf{Method}

- \texttt{right-to-left-spline-connect Left-object Right-object} of MRE-G:Connection \hspace{1cm} \textbf{Method}

- \texttt{spline Object1 Object2} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  Here no choice is provided for the programmer, the objects are connected totally
  as a function of their relative orientation.

- \texttt{straight Object1 Object2} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  A "t-bar" connection is made between the objects. This implies that 3 straight
  line must be drawn.

- \texttt{top-to-bottom-connect top-object bottom-object} of MRE-G:Connection \hspace{1cm} \textbf{Method}

- \texttt{top-to-bottom-spline-connect Top-object Bottom-object} of MRE-G:Connection \hspace{1cm} \textbf{Method}

- \texttt{vertical-connect} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  Connects Object1's top or bottom entry point to Object2's top or bottom entry
  point.

- \texttt{Vertical-spline-connect Object1 Object2} of MRE-G:Connection \hspace{1cm} \textbf{Method}
  These reduce to one degree of freedom the choice of which spline connect to
  use.
The Drawing-Mixin Flavor

The *Drawing-Mixin* is a primary, internal mixin which is mixed into objects which need to draw themselves and are transformable.

Instance Variables

The following are instance variables in the *Drawing-Mixin*:

- (Visible? nil)
  This is non-nil when the object is visible on the screen.

Messages

The following are messages to the *Drawing-Mixin*:

- **:draw optional**
  \[
  (x-to \ x) \ \\
  (y-to \ y) \ of \ MRE-G:*flavor* \]
  This draws the object at point X-To Y-To.

- **:draw-spline \ x \ y** of MRE-G:*flavor*
  Here \( x \) and \( y \) are arrays of points which serve to control a bezier curve.

- **:erase** of MRE-G:*flavor*
  Erases the object.

- **:move \ x \ y** of MRE-G:*flavor*
  Updates the objects X and Y coordinates properly.
The Dynamic-Placement-Mixin Flavor

The Dynamic-Placement-Mixin flavor is mixed into the Graphics-Space flavor. This is intended to give the experienced user a system programmer like control over the placement mechanism. This flavor has the following instance variables:

- (Ring-Topology-Used? nil)
  This is set to a non-nil value if the user send a :Connect-Objects message to a graphics space instance with a :Placement key of (:Ring-Topology). When this is non-nil the entry points of all nodes become floating entry points (See Glossary). The actual entry point appearance is changed only upon connection refresh.

- (max-number-of-trials-for-swing-placement nil)
  This controls the number of trials used when placing nodes by iterative technique. This may be set using the:
  (:set-max-number-of-trials-for-swing-placement n) message to the graphics-space instance. After the allotted n tries the placement algorithm gives up and leaves the node at its last position. When placing iteratively tree skew is to be expected. Thus because of the time penility due to iterative placement activity and because of the tree skew it is expected that iterative placement will be used sparingly. It is recommended that a multi-growth-tree connected to a normal tree inhibit placement until an application program has stopped creating nodes (that is placement occurs at a logical point). The node will look for free space on the same level as its siblings. The node looks within the constraints of the :Placement flag and the :Max-Number-Of-Trials-For-Swing-Placement in addition to the :Placement-Conflict-Criterion. After all this stuff comes into play some nodes will not be able to be placed on the same level as there siblings. When this happens the level constraint is relaxed and the next level is sought. Thus after :Max-Number-Of-Trials-For-Swing-Placement**2 times placement may still fail and the node will be left in a bad place...at this point the user may choose to increase the :Max-Number-Of-Trials-For-Swing-Placement or try to place the node with an external application program or by hand using the mouse.

Messages To The Dynamic-Placement-Mixin

Since this is only mixed into the graphics space all the following messages are available to users of instance of Graphics-Space flavors.

- :After :Connect-Objects
  
  obj1 obj2
  &allow-other-keys
  &key
  
  (placement-conflict-criterion :objects-avoid-objects)
  (entry-points :discrete)
  (level? t) ;; if this is t all nodes must be level with there siblings.
  (place-now? t) ;; If this is nil the user starts the placement by hand.
  (placement :Ring-Topology) ;; connecting-angle is also available.

  of MRE-G:Dynamic-Placement-Mixin

  These are documented more fully in Messages To Graphics-Spaces.
The Graphics-Space Flavor

"Graphics-space" is a flavor which is created by the Create-Graphics-Space function. Certain instance variables are useful to the user and are documented here:

- (Object-stack nil)
  This is a list of all known objects in the graphics-space instance.

- (map-to-window terminal-io)
  This is usually initialized when the create-graphics-space function is invoked. The instance variable is usually refered to but not set.

- :Focus-On-Active-Object &Key :Switch <Flag> of MRE-G:Graphics-Space  Method
  This will cause the most recently connected child node to be centered in the graphics space if the child is drawn out of sight. The user should note that the graph generation will be slowed by the scrolling required. Flag may be :on or :off.

- :Clear of MRE-G:Graphics-Space  Method
  This will clear a graphics space and all related objects.

- :Scroll XY of MRE-G:Graphics-Space  Method
  Causes the graphic space to scroll the indicated amounts.

Messages

The following are messages to the graphics-space:

- :ADD-MOUSING-MESSAGE of MRE-G:Graphics-Space  Method
  &key message
  
  menu-name
  documentation
  (default-p nil)

  ° :Message
  This is the message which will be sent to the object on the screen.

  ° :Documentation
  This is the documentation string which will appear in the who line.

  ° :Menu-Name
  This is the string which will appear in the menu.

  ° Default-p
  When this is non-nil the message, menu-name and documentation are bound to mouse-l.

- :CLEAR &key (window t) of MRE-G:Graphics-Space  Method
Object and connection records are erased and if *window* non-nil window is cleared.

- **:CONNECTION-STACK** of MRE-G:Graphics-Space
  Returns a list of all connections made in the graphics space.

- **:CREATE-GLOBAL-DEFAULT-MENU** of MRE-G:Graphics-Space
  Returns an instance of the pop-up menu used when the mouse is clicked left. This is called once upon graphics space initialization.

- **:DEFAULT-MENU** of MRE-G:Graphics-Space
  Installs a series of menu items which appear when the user clicks right on a mouse sensitive item in the graphics-pane. This also serves to make all nodes update there extent box listings in the window for highlighting purposes.

- **:DEFAULT-MOUSE-HANDLER-FOR-NON-NODES** blip of MRE-G:Graphics-Space
  If the user clicks the mouse over a non-node this method is called with the original mouse blip. The blip is sure to be a list and its first element to be `mouse-button`.

- **:DRAW-CONNECTIONS** of MRE-G:Graphics-Space
  Sends all connections in the graphics-space an `:draw` message.

- **:DRAW-OBJECTS** of MRE-G:Graphics-Space
  Sends all objects in the graphics-space an `:draw` message.

- **:ERASE-CONNECTIONS** of MRE-G:Graphics-Space
  Sends all connections in the graphics-space an `:erase` message.

- **:ERASE-OBJECTS** of MRE-G:Graphics-Space
  Sends all objects in the graphics-space an `:erase` message.

- **:GREATEST-GRANDCHILDREN** of MRE-G:Graphics-Space
  This returns the leaves of a family oriented tree.

- **:HARDCOPY** of MRE-G:Graphics-Space
  Makes laser copy of `:Map-to-window`.

- **:OBJECT-STACK** of MRE-G:Graphics-Space
  Returns a list of all objects made in the graphics space.

- **:ORPHANS** of MRE-G:Graphics-Space
  Returns a list of all objects made in the graphics space with no parents.

- **:UPDATE-OBJECT-BORDERS-AND-MESSAGES** of MRE-G:Graphics-Space
  Despite name this message only updates the extent boxes in the *item-type-alist* in the graphics pane for highlighting purposes.
The Intersection-Grid-Mixin Flavor

This flavor is mixed into the Graphics-Space flavor. This gives the user the flexibility to do node-node and node-connection intersection calculation.

Messages Added To Graphic Spaces

- \( :\text{LineIntersectionP} \langle\text{StartX}\rangle \langle\text{StartY}\rangle \langle\text{EndX}\rangle \langle\text{EndY}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a non-NIL value if any node intersects along the indicated line.

- \( :\text{NodesAtPoint} \langle\text{x}\rangle \langle\text{y}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a list of all the nodes at the point.

- \( :\text{NodesForLineIntersection} \langle\text{StartX}\rangle \langle\text{StartY}\rangle \langle\text{StartX}\rangle \langle\text{EndY}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns all the nodes which intersect along the specified line.

- \( :\text{NodesInRectangle} \langle\text{UpperLeftX}\rangle \langle\text{UpperLeftY}\rangle \langle\text{LowerLeftY}\rangle \langle\text{UpperRightX}\rangle \&\text{Optional} \langle\text{NodesToSkip}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a list of nodes which are intersecting with the specified rectangle.

- \( :\text{NodesInRectangleP} \langle\text{UpperLeftX}\rangle \langle\text{UpperLeftY}\rangle \langle\text{LowerLeftY}\rangle \langle\text{UpperRightX}\rangle \&\text{Optional} \langle\text{NodesToSkip}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a non-NIL value if there are any nodes within the rectangle.

- \( :\text{PointIntersectionP} \langle\text{x}\rangle \langle\text{y}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a non-NIL value if there is any node at a point.

- \( :\text{SendForNodesInRectangle} \langle\text{Message}\rangle \langle\text{UpperLeftX}\rangle \langle\text{UpperLeftY}\rangle \langle\text{LowerLeftY}\rangle \langle\text{UpperRightX}\rangle \&\text{Optional} \langle\text{NodesToSkip}\rangle \) of MRE-G:Intersection-Grid-Mixin
  :\text{ObjectsWhichIntersect}

Messages Added To Nodes

- \( :\text{Intersects} \) of MRE-G:Intersection-Grid-Mixin
  Returns a non-NIL value if any node intersects with the current placement of this node.

- \( :\text{MapForIntersectedNodes} \langle\text{x}\rangle \langle\text{y}\rangle \langle\text{Fn}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Applies (FUNCTIONs) a function to each node that would intersect with the node if it were moved to the specified position, other than itself.

- \( :\text{NodeIntersectionP} \langle\text{x}\rangle \langle\text{y}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a non-NIL value if the node would intersect with any other node, other than itself, if it were moved to the indicated position.

- \( :\text{NodesInIntersection} \langle\text{x}\rangle \langle\text{y}\rangle \) of MRE-G:Intersection-Grid-Mixin
  Returns a list of nodes that would be intersected if the node were moved to the
indicated position, other than itself.

- **:ObjectsWhichIntersect of MRE-G:Intersection-Grid-Mixin**
  Returns a list of all nodes which intersect with this node.

- **:RectangleOverlapsP <UpperLeftX> <UpperLeftY> <LowerLeftY> <UpperRightX> of MRE-G:Intersection-Grid-Mixin**
  Returns a non-NIL value if the specified rectangle overlaps any portion or all of the node.

- **:SendForIntersectedNodes <x> <y> <Message> of MRE-G:Intersection-Grid-Mixin**
  Sends a message to each node that would intersect with the node if it were moved to the specified position, other than itself.
The Kinetic-Mixin Flavor

The Kinetic-Mixin is mixed into the Node-Mixin. The Kinetic-Mixin is designed to give a centralized location for methods which deal with animating an object (as opposed to basic object drawing primitives like move and draw).

Mixins

The Kinetic-Mixin has the Calculation-Mixin mixed in.

Instance Variables

The Kinetic-Mixin has one settable instance variable:

- (angular-position 0)
  This is an angle in degrees which is used to show the relative rotation of an object with respect to its' initial orientation.

The Kinetic-Mixin handles the following messages:

- :absolute-rotation theta of MRE-G:Kinetic-Mixin
  Orients the object theta degrees from its' initial orientation.

- :family-follow-mouse of MRE-G:Kinetic-Mixin
  Causes the entire family to be dragged by the mouse.

- :follow-mouse of MRE-G:Kinetic-Mixin
  Causes the object to follow the mouse as long as the mouse button is held.

- :slide x-to y-to of MRE-G:Kinetic-Mixin
  Causes the object to move from present location to x-to y-to gradually.

- :slide-along-connection from-object to-object connection of MRE-G:Kinetic-Mixin
  Causes an object to move from the from-object to the to-object along the connection. This does not work for splined or T connections.

- :spin omega &optional x y of MRE-G:Kinetic-Mixin
  The object spins about x y for omega degrees. This message is only for transformable objects (like the arrow, line or fabricated object).

- :spin-about-mouse omega of MRE-G:Kinetic-Mixin
  Allows the object to spin about the mouse.
The Node-Mixin Flavor

The node-mixin is a required flavor for all nodes in the graphics space.

Mixins

The node-mixin has:

- Node-Placement-Mixin
  - Debugging-Mixin
  - Copyable-Property-List-Mixin
  - Basic-Copyable-Object
  - Mousable-Mixin and
  - Kinetic-Mixin
  - mixed in.

Instance Variables

The node-mixin has the following instance variables:

- (graphics-space nil)
  This is a pointer to the graphics-space in which the object resides. Only one graphics-space is permitted per object. (parents nil)

- This is a list of parents.

- (object-description nil)
  Set by graphics space upon creation.

- (maximum-downward-connecting-angle 90)
  (maximum-leftward-connecting-angle 75)
  (maximum-rightward-connecting-angle 75)
  (maximum-upward-connecting-angle 75)
  All in degrees, these angles are the max angles of spread between children.

- (left-extent nil)
  (right-extent nil)
  (top-extent nil)
  (bottom-extent nil)
  Extents define the outer boundary of the mouse sensitive boxes. Send the message :Calculate-Extents to properly initialize.

- (x-left-entry-point nil)
  (y-left-entry-point nil)
  (x-top-entry-point nil)
  (y-top-entry-point nil)
  (x-right-entry-point nil)
  (y-right-entry-point nil)
  (x-bottom-entry-point nil)
(y-bottom-entry-point nil)
Entry-points are used to define where the connection will make contact with the node. send the message :Calculate-Entry-Points to properly initialize.

- (inter-block-gap 5)
  This is used to calculate the Extent box.

- (children nil)
  This is a list of all children to the node.

- (arrows-which-point-to-me nil)
  These are all the arrow instances which are supposed to point towards a node.

- (font fonts:cptfont)
  This is the font for the string of this node.

- (window terminal-io)
  This is the window of the node.

- (string nil)
  This is the nodes label.

- (place-your-children-flag :down)
  This may take on the values :up :left :right or :down.

- (connection-store nil)
  This is a list of connections to the object.

- (highlight-flag :off)
  This is always :on or :off and is usually set by the :switch message.

**MRE-G:Node-Mixin flavor**

The **Node-Mixin** handles the following messages:

- **:add-arrow-which-points-to-you arrow-instance** of MRE-G:Node-Mixin
  This stores an arrow instance in the arrows-which-point-to-me instance variable.

- **:add-child child** of MRE-G:Node-Mixin
  Adds the child to the list of children.

- **:add-connection connection** of MRE-G:Node-Mixin
  Adds the connection instance to the connection store.

- **:adjacent-nodes** of MRE-G:Node-Mixin
  Returns a list of all nodes connected to the node.

- **:all-ancestors** of MRE-G:Node-Mixin
  Returns a list of all ancestors.

- **:all-descendants** of MRE-G:Node-Mixin
Returns a list of all descendants.

  Tests the new member to see if cycles are introduced into the graph.

- :bottomists-at-level-n n of MRE-G:Node-Mixin
  Method

- :center of MRE-G:Node-Mixin
  Centers the node. Provisions are available for expansion but are not yet ready.

- :center-self-unconditionally of MRE-G:Node-Mixin
  Always centers the node.

- :children-at-level-n n of MRE-G:Node-Mixin
  Method
  Returns a list of all children at generation n.

- :connect-objects of MRE-G:Node-Mixin
  Method
  Used when interacting with the screen. The user must send the other object message to the object which is to be connected to.

- :copy-family of MRE-G:Node-Mixin
  Method
  Copies the entire family of nodes with a fixed offset.

- :copy-node of MRE-G:Node-Mixin
  Creates a new node of similar type in the graphics-space.

- :delete of MRE-G:Node-Mixin
  Method
  Deletes the node and all connections to the node.

- :delete-child child of MRE-G:Node-Mixin
  Method
  Removes the child from the children list.

- :delete-connection connection of MRE-G:Node-Mixin
  Method
  Deletes the connection instance from the connection list.

- :distance-from-mom <node> of MRE-G:Node-Mixin
  Method
  Returns the distance in pixels <node>.

- :distance-from-point node x y of MRE-G:Node-Mixin
  Method
  returns the distance from node to point x y.

- :draw-all-arrows-to-children of MRE-G:Node-Mixin
  Method
  Propagates the draw-arrows-to-children to all descendants.

- :draw-all-connections of MRE-G:Node-Mixin
  Method
  Draws connections for entire family.

- :draw-arrows-to-children of MRE-G:Node-Mixin
  Method
  This causes all the connection drawn to children to have arrows which point to the children.
• **:draw-connections** of MRE-G:Node-Mixin sends the :draw message to every connection in the connection store.

• **:erase-all-connections** of MRE-G:Node-Mixin erases connection in family.

• **:erase-connections** of MRE-G:Node-Mixin sends an erase message to all connections in the connection-store.

• **:family** of MRE-G:Node-Mixin
  Returns a list of all members of the family.

• **:family-extent arg** of MRE-G:Node-Mixin
  arg may be :left, :right, :top, or :bottom.

• **:family-node-nearest-point x y** of MRE-G:Node-Mixin
  Returns the node in the family which is nearest to the given point.

• **:grandchildren** of MRE-G:Node-Mixin
  Returns a list of level 2 descendents.

• **:greatest-grandchildren** of MRE-G:Node-Mixin
  Returns a list of the deepest descendents.

• **:height-of-children-at-level-n n** of MRE-G:Node-Mixin
  Returns the pixel height of the children at level n.

• **:how-many-children-at-level-n n** of MRE-G:Node-Mixin
  Returns the number of children at this level of descendency.

• **:inside-bottom-extent** of MRE-G:Node-Mixin
  Returns extent values in pixels.

• **:inside-left-extent** of MRE-G:Node-Mixin

• **:inside-right-extent** of MRE-G:Node-Mixin

• **:inside-top-extent** of MRE-G:Node-Mixin

• **:largest-outside-dimension** of MRE-G:Node-Mixin
  returns the larger: outside width or height.

• **:leftists-at-level-n n** of MRE-G:Node-Mixin

• **:make-visible** of MRE-G:Node-Mixin
  decides if object is visible and places itself in center if it is not.

• **:maximum-number-of-generations** of MRE-G:Node-Mixin
  Returns the number of levels of descendency.

• **:minimum-distance-from-the-mother-node** of MRE-G:Node-Mixin
Returns the distance from the closest child.

- **:minimum-height-of-all-children** of MRE-G:Node-Mixin
  Returns sum of all the tallest children for each generation.

- **:move-children dx dy** of MRE-G:Node-Mixin
  Moves only children.

- **:move-family dx dy** of MRE-G:Node-Mixin
  Does a relative move on all members of the family.

- **:nearest-child** of MRE-G:Node-Mixin
  Returns the nearest child instance.

- **:orphans** of MRE-G:Node-Mixin
  Returns a list of all orphans in the family.

- **:other-object** of MRE-G:Node-Mixin
  Returns the object instance. This is used in conjunction with the
  :connect-objects method.

- **:outside-height** of MRE-G:Node-Mixin

- **:outside-width** of MRE-G:Node-Mixin
  Uses extents to calculate values.

- **:place-children &key :growth <flag>** of MRE-G:Node-Mixin
  *Flag* may be :left, :right, :up or :down.

- **:place-children-left** of MRE-G:Node-Mixin
  traverses the family placing the children to the left.

- **:play-family** of MRE-G:Node-Mixin
  Plays all objects in family. This only works on machines with the sound option.

- **:rightists-at-level-<n>** of MRE-G:Node-Mixin
  These are the nodes which are below, right, left (or are above) at a specific
  generation away from the node recieving the message.

- **:scroll** of MRE-G:Node-Mixin
  orphans in the graphics-space.

- **:shrink &optional percentage** of MRE-G:Node-Mixin
  Shrinks by a percentage. 0 < percentage < 1.

- **:shrink-family** of MRE-G:Node-Mixin
  Stores the present label and reduces the node-size. Use this only once since the
  label will be lost after 2 shrinks.

- **:squish-down** of MRE-G:Node-Mixin
• :squish-left of MRE-G:Node-Mixin Method
• :squish-right of MRE-G:Node-Mixin Method
• :squish-up of MRE-G:Node-Mixin Method
• :switch &key :Highlight <Flag> of MRE-G:Node-Mixin Method
  Increases the border size for the node.
• :tallest-child of MRE-G:Node-Mixin Method
  Returns a child instance or nil.
• :tallest-child-in-generation-n n of MRE-G:Node-Mixin Method
  Returns the child instance which is the tallest for the generation.
• :to-the-bottom? node of MRE-G:Node-Mixin Method
• :to-the-left? node of MRE-G:Node-Mixin Method
  If node is to the left this returns non-nil.
• :to-the-right? node of MRE-G:Node-Mixin Method
  Tests to see if node is to-the-(right or below) and returns non-nil if it is.
• :to-the-up? node of MRE-G:Node-Mixin Method
  If node is above this returns non-nil.
• :topists-at-level-n n of MRE-G:Node-Mixin Method
• :total-refresh of MRE-G:Node-Mixin Method
  Sends a refresh to the graphics-space.
• :unshrink of MRE-G:Node-Mixin Method
  Sets the node back to its original size.
• :viewable? of MRE-G:Node-Mixin Method
  Returns t if object is viewable. This can be fooled by sending a refresh to the
  graphics-pane, if this happens a :refresh sent to the graphics-space may clear
  things up.
• :width-of-children-at-level-n n of MRE-G:Node-Mixin Method
  Returns the pixel width of the children at level n.
• :window-extent of MRE-G:Node-Mixin Method
  Side may be :left, :right, :top, :bottom, :midpoint-x or :midpoint-y. Values
  returned are in pixels.
Well these messages do placement by compacting existing placement. They
propagate through the family descendants. Current thinking about placement is
that we should have messages which do the same thing but with a limited
scope, this would allow for more flexible topological representations.

Node-Mixin Flavor Mixins
The following are mixed into the Node-Mixin flavor:
*Node-Placement-Mixin* and the *Debugging-Mixin*.
The String Flavor

"String" is a flavor with the following instance variables:

- (x 100)
- (y 100)
- (x2 200)
- (y2 200)
- (visible? nil)
- (character-length-limit :none)
- (stashed-string ")"
- (string ")"
- (font fonts:cptfont)
- (window terminal-io)

The following may be set upon instance variable creation:

- :x, :y
  The start points of the string.

- :x2, :y2
  The end points of the string.

- :font
  The font of the string must be any currently loaded font.

- :window
  The window (with a tv:graphics-mixin) in which the string appears. To reset any of the above variables the following methods exist. If a method does not exist for changing the property of your choice (such as the :font) the user is advised to erase the string first.

Methods Of The String Flavor

- :character-trim-string of MRE-G:String
  Used internally to reduce the string length and append a "." when truncation occurs.

- :draw &optional new-x new-y new-x2 new-y2 of MRE-G:String
  Draws the string from new-x new-y towards new-x2 new-y2. If the string does
not fit it is stretched. Very little compression is possible. If new-x is present new-y must be present also. If new-x2 is present then so must new-y2.

- **:erase** of MRE-G:String
  Erases string by using the exclusive-or alu function.

- **:follow-mouse** of MRE-G:String
  Mouse-I must be held down and in motion before this message will work. String will be drawn towards the mouse as long as mouse-I is held. While the kinetic-mixin is mixed in, the string does not support the messages present and the user is discouraged from trying them.

- **:move new-x new-y &optional new-x2 new-y2** of MRE-G:String
  Makes new-x, new-y the new start position for string. If new-x2 is present then so must new-y2.

  - **:new-string string** of MRE-G:String
    The drawn string takes on the shape of the new string and is drawn. The character-length-limit is reset to :none. This may be used regardless of visibility status. :length length &optional (in-pixels? nil) The drawn string is shortened of lengthed to the character-length if in-pixels? is nil. If in-pixels? is on-nil then the length is taken to be in pixels.

- **:pixel-height** of MRE-G:String
  returns pixel height of string.

- **:pixel-length** of MRE-G:String
  returns true pixel length (unstretched) of string.

- **:restore-old-string** of MRE-G:String
  used internally whenever the length is changed.

- **:save-string** of MRE-G:String
  used internally to save the string whenever a :new-string message is sent.

- **:trim-string** of MRE-G:String
  used internally to reduce the size of the string by using :character-trim-string if needed.
The Transformable-Graphics-Object-Mixin Flavor

This provides homogenous coordinate transforms for a transformable graphics object.

Mixins

The following flavors are mixed in:
calculation-mixin and drawing-mixin.

Instance Variables

The following settable instance variables are provided:

- \( px \) \( py \)
  These are arrays of points which are to be transformed.

- \( \text{rotation} \ 0 \)
  This is the angular displacement in degrees of the object.

- \( \text{number-of-points} \ 0 \)
  This is the number of points in \( px \) and \( py \).

Required Methods

The following are code requirements:

- \( :\text{make-array} \)
  This fills \( px \) and \( py \) and sets number-of-points.

Messages

The following are valid messages:

- \( :\text{origin-rotate theta} \)
  This rotates the object about the origin by \( theta \) degrees.

\[ :\text{rotate \&key} \]

\( (\text{theta-z} \ 0) \)
\( (\text{x-center} \ (\text{send self} \ :x)) \)
\( (\text{y-center} \ (\text{send self} \ :y)) \)
\( (\text{absolute nil}) \) of MRE-G:Transformable-Graphics-Object-Mixin

This rotates the object \( \text{theta-z} \) degrees about \( \text{x-center} \ \text{y-center} \). If absolute is non-nil the rotation is not relative. \( :\text{after \ :init} \)

of MRE-G:Transformable-Graphics-Object-Mixin

The \( :\text{make-array} \) message is sent.

- \( :\text{initialize-orientation \&key (theta-z 0)} \)
of MRE-G:Transformable-Graphics-Object-Mixin

Theta-z is the angle about the z axis (out of the screen) in degrees to set the
objects orientation.

- **:move dx dy** of MRE-G:Transformable-Graphics-Object-Mixin
  Does a relative translation.

- **:scale &optional**
  
  (sx 1.)
  (sy sx) of MRE-G:Transformable-Graphics-Object-Mixin

  This scales the object relative to current size.

- **:translate &optional (dx 0) (dy 0)** of MRE-G:Transformable-Graphics-Object-Mixin

  Does a relative translation.
Creating Your Own Node Shapes

One day the experienced user will get tired of looking at circles and rectangles. The user would like to create a custom node shape and desires more flexibility then the Fabricated-Object can afford.

Code Requirements

When the user whishes to create a custom node shape certain requirements must be met:

1. All required instance variables must be present in the custom flavor.
   - \textit{x}
   - \textit{y}
     X and Y coordinates for the center of the object. Defaults are required and 500 500 is recommended.
   - \textit{class}
     A colon followed immediately by an atom must be present at default.

2. All required methods must be present in the custom flavor.
   - \textit{:adjust-size-for-string}
     This will adjust the size of the object probably by using the \textit{:pixel-length} message.
   - \textit{:draw \&optional ignore ignore} ...
     This will draw the object by sending the instance variable window draw messages. The center of the object will be \textit{X} and \textit{Y}. The first 2 ignores are used by a before demon to set up the center coordinates. The ... in the lambda-list is used to indicate more information the user might like to add.
   - \textit{:inside-height}
     Must return the inside height in pixels.
   - \textit{:inside-width}
     Must return the inside width in pixels.

3. The flavor's file must have the \textit{MRE-G} package in its attribute list.

4. The flavor must have settable instance variables.

5. The flavor must require and node-mixin variables it intends to use, including but not limited to, the window. The window will almost always be required if drawing is intended, the only exception would be if the user obtained the window instance by the form:
6. The flavor must have the node-mixin flavor mixed-in.

In order to provide a clearer explanation I shall sit the Circle node as an example:

Creating The Circle Flavor

What follows is the actual MRE-Graphics circle flavor. This is shown in order to give the user an idea of what is involved in defining a new node type. The file which defines the flavor must have the MRE-Graphics package in its’ attribute list.

(send self :window)

(defun circle (x 500)  
  (y 500)  
  (radius 30)  
  (class 'circle))  
  (node-mixin)  
  (:required-instance-variables window)  
  (:settable-instance-variables)

(defun circle :adjust-size-for-string ()
  (setq radius (+ 4 (send self :pixel-length) 2)))  
  (if (< radius 4) (setq radius 4)))

(defun circle :draw) (optional ignore ignore r)
  (if r (setq radius r))
  (send window :draw-circle x y radius tv:alu-xor)
  (if (eq (send self highlight-flag) :on)
      (send window :draw-circle x y (- radius 1) tv:alu-xor)))

(defun circle :inside-height ()
  (* 2 radius))

(defun circle :inside-width ()
  (* 2 radius))

Note how the :draw message provides code for the highlight mode. This is optional and if it is omitted highlight will simply do nothing. Also, see how the radius parameter was “tacked” on so that an external message could be sent to change radius without having to erase the node and send a set message.
Glossary

- **Child** A node decedent from a parent. It is the leaf of a subtree but could become a parent to another node.

- **Entry-Points** This is the place on a node which makes contact with a connection.

- **Extents** A node is approximated by a rectangle which usually encompasses the node. This rectangle marks the outside node extents. The extents are used in object intersection calculation and are shown when mouse sensitive objects are within proximity to the mouse. The :Inside-Width and :Inside-Height methods (See Creating Your Own Node Shapes) are used to calculate extents.

- **Family** A list of nodes which contains parents and there children.

- **Floating-Entry-Points** These are entry-points which move continously as the object moves. They always pick the point on an object which is closest to ring-topological center.

- **Parent** An ancestor node which could have existed before any of its descendants. It is the origin in a tree or subtree. This could be a child to another node.

- **Ring-topology** This is a placement which is patterned after a circle. All nodes in this pattern fall on the rim of the circle and entry point are usually floating in order to reduce connection-node interference.