The GNOME Canvas

by Federico Mena Quintero
<federico@nuclecu.unam.mx>

The GNOME canvas is an engine for structured graphics that offers a rich imaging model, high performance rendering, and a powerful, high—level API. It offers a choice of two rendering back-ends, one based on Xlib for extremely fast display, and another based on Libart, a sophisticated, antialiased, alpha-compositing engine. Applications have a choice between the Xlib imaging model or a superset of the PostScript imaging model, depending on the level of graphic sophistication required.

This white paper presents the architecture of the GNOME canvas and Libart, and describes the situations in which it is useful to use these technologies.

Introduction

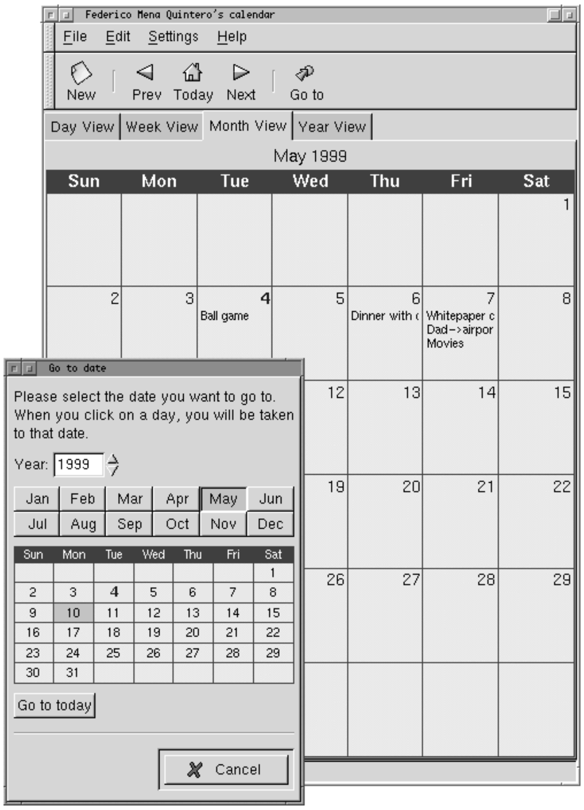
The GNOME canvas is a high-level engine for creating structured graphics. This means the programmer can insert graphical items like lines, rectangles, and text into the canvas, and refer to them later for further manipulation. The programmer does not need to worry about repainting these items or generating events for them; the canvas automatically takes care of these operations.

The canvas provides several predefined item types, including lines, rectangles, and text. However, the canvas is designed to serve as a general-purpose display engine. Applications can define their own custom item types that integrate with the rest of the canvas framework. This lets them have a flicker-free display engine with proper event management and propagation.

The canvas supports two rendering models. One is based directly on Xlib (used via GDK), which provides an extremely fast and lean display that runs well over networks. The second rendering model is based on Libart, a sophisticated library for manipulating and rendering vector paths using antialiasing and alpha compositing. Libart provides a superset of the PostScript imaging model, allowing for extremely high-quality displays.

Simple applications can use the predefined canvas item types to create interactive graphics displays. For example, the GNOME Calendar program uses the canvas to display and manipulate monthly calendars. It only needs simple graphical items like rectangles and text to display its information. Please look at Figure 1 for an example of the use of the canvas in the GNOME Calendar.

Figure 1. Use of the canvas in the GNOME Calendar

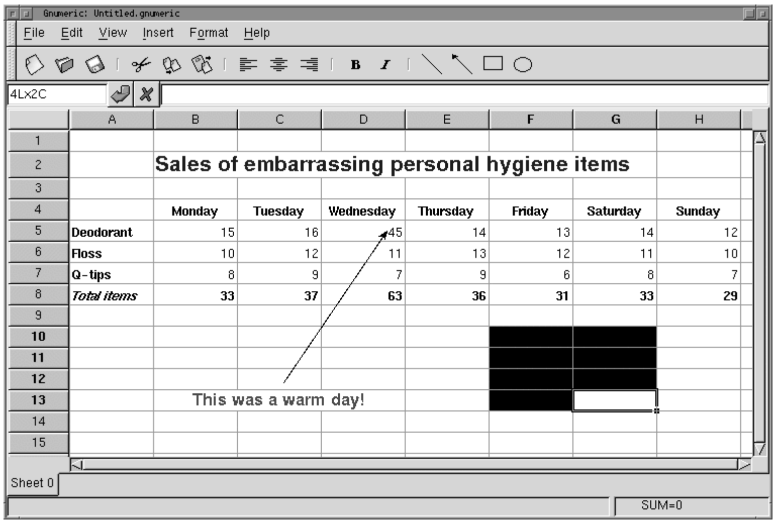


(1) The whole month view is a single big canvas. Stock canvas items like rectangles and text are used to display an easily-customizable calendar.

(2) The little monthly calendars are custom canvas groups based on the stock item types. The calendar program implements different behavior for each place in which the monthly calendar item is used.

Applications with more sophisticated requirements can define their custom canvas item types. The Gnumeric spreadsheet defines a large 'Sheet' item that takes care of painting the sheet's grid and the cell contents. It also defines an item to handle the complex cursor object in the spreadsheet, which must handle the current selection, the current cell, and the border items to drag and extend selections. Please look at Figure 2 for an example of the use of the canvas in the Gnumeric spreadsheet.

Figure 2. Use of the canvas in the Gnumeric spreadsheet



(1) Gnumeric uses a custom "Sheet" item to display the spreadsheet contents. This item is responsible for drawing grid lines and cell contents.

(2) The cursor is another custom item. In Figure 2 it is shown as a 4×2-cell selection with the current cell being G13. The cursor item handles all aspects of a spreadsheet's cursor, including the current cell, the selection range, and the clickable areas that let the user drag and fill cells automatically.

(3) The column and row headers are two modes of a single custom item. This item draws the button-like headers and highlights them as appropriate when cells are selected.

(4) Gnumeric uses the stock canvas items (like lines, rectangles, and ellipses) for the graphic elements that the user may want to overlay on the spreadsheet. Here the user has inserted an arrow, and the canvas takes care of repainting everything automatically.

This white paper describes the architecture of the GNOME canvas and Libart. It gives examples on why application writers may want to use these technologies.

The Canvas Architecture

The original version of the GNOME canvas was based on the canvas widget from the Tk toolkit. From the standpoint of the application programmer, the canvas presents the following characteristics:

- A canvas appears as a normal GTK+ widget with its own GDK window (or X window).

- The programmer can insert graphical items into the canvas. The canvas provides several predefined item types, including lines, rectangles, ellipses, polygons, and text. Canvas items can be manipulated after they are created and inserted into the canvas. Common operations include changing the color of an item or moving it to a different position.
- Canvas items receive events just as if they were normal X windows or other widgets, and the programmer can bind these events to actions. Common actions include moving an item when the user drags it with the mouse, or changing an item's color when the mouse enters its visible area.
- Canvas items are normal GTK+ objects. Custom item types can be created by simply deriving a new object class from the base GnomeCanvasItemClass. Items emit events in the form of GTK+ signals, just like other widgets.
- The canvas takes care of all drawing operations so that it never flickers, and so that the user does not have to worry about repainting the items he or she wants to display.

The canvas allows for hierarchical drawings by using nested groups of items. It uses recursive bounding boxes that allow culling of items for efficient repainting. Operations such as moving or deleting a group apply to all the items inside the group. This makes it easy to create and manipulate hierarchical graphics displays.

Canvas Items

Canvas items are normal GTK+ objects. All canvas items are ultimately derived from an abstract GnomeCanvasItemClass that defines the basic operations all canvas items must provide, like painting the item or deciding whether a point is inside it. Using the GTK+ object system provides several advantages:

- No extra work is involved in wrapping canvas items for different language bindings, so the canvas is usable from languages like Scheme, Perl, Python, and C++.
- Canvas items use the GTK+ signal/slot mechanism to emit events, making it easy to define behavior based on the events that items receive.
- One can associate arbitrary data items to canvas items by using the GTK+ dataset mechanism.

All the attributes of items (line style, color, position) are configured using the GTK+ object argument mechanism. Since items may have many configurable attributes, using the object argument mechanism allows us to minimize the number of API entry points, and also makes it easier to create language bindings for the canvas.

Items can be hidden and shown at any time, as well as modified using affine transformations. An affine transformation is a mapping of the plane onto itself, specified as a 3×3 matrix that can be multiplied by a vector that specifies a 2D point to transform it to a different coordinate system. The user can specify an arbitrary affine transformation matrix to be applied to an item. Convenience functions are provided for common operations like translating, scaling, and rotating and item.

Grouping of Items

Items can be organized in the canvas using a tree hierarchy. Items can be groups (nodes in the tree), or terminal items (leaves in the tree). Groups can contain any number of children, which can be leaf items or other groups. A GnomeCanvasGroupClass is provided to manage groups of items. Items can be nested to an arbitrary depth inside the canvas.

A canvas has a single root group. Simple drawings or diagrams can be created by inserting all leaf items directly under the root. Complex schematics and hierarchical drawings can be created by nesting groups together. For example, a circuit editor may use small groups of basic items to create logic gates. More complex components could be created by combining the groups that represent logic gates, and complete circuits could, in turn, be composed of these components.

Operations on a group apply to all of its children; for example, moving a group produces the same visual effect as moving each child individually.

A canvas item must know how to compute the distance between a point and itself, so that the canvas can tell whether the mouse is inside an item or not. For efficiency, items keep a rectangular bounding box that lets the canvas ignore an item if a point is tested to be outside the item's bounds, which can be done very quickly. In turn, a canvas group will make its bounding box big enough to encompass all of its children's bounding boxes, allowing for efficient recursive culling of items.

Items inside a group are stacked on top of each other, and items that are higher up in the stack obscure the items below them. An item can be raised or lowered in its parent group's stack.

Behavior of Items

The canvas does not define any default behavior for items. Instead, the programmer can create signal connections to the event signal in items and define the appropriate behavior.

Items get the normal user-initiated events, such as mouse button press/release events, mouse motion events, mouse enter/leave events, key press/release events, and focus in/out events.

When an event signal is emitted for an item, it is propagated upwards in the item hierarchy until it is marked as handled by one of the event handlers. This works in the same way as event propagation in the GTK+ widget system.

The Updating and Rendering Process

The canvas uses an update/render process when something requires a change in appearance. This process goes as follows:

1. A state change happens in a canvas item, usually from direct manipulation through the user interface.
2. The canvas item requests an update from the canvas. The item is thus marked as "requiring an update". The canvas installs an idle handler on the GTK+ main loop.
3. The application keeps running, possibly requesting updates for other items, until it gets back to the GTK+ main loop. This is where idle handlers are run.
4. The idle handler for the canvas is run. Here, the canvas calls the update method of each item that requested an update. The update method may then queue a

redraw of a certain area of the item. This area is represented as a microtile array, to be described later.

5. The canvas decomposes the final microtile array into a list of rectangles that need repainting.
6. The canvas calls the draw or render method of each item that intersects one of these rectangles, depending on whether the canvas is in Xlib or Libart modes, respectively.
7. The canvas is now fully updated and redrawn, and the application continues running.

This method has some important characteristics. First, all updates and redraw requests are delayed until the idle loop. This ensures that the canvas will not try to repaint itself until the last state change to an item has happened, and may also reduce the number of update-related operations that need to be performed. For example, changing the coordinates of a polygon's vertices several times is equivalent to changing them just once to the final position. Also, items are asked to draw themselves onto a temporary buffer (a GDK pixmap in the case of the Xlib renderer, or an RGB buffer in the case of the Libart renderer). This completely eliminates flicker.