The GNOME Canvas: a Generic Engine for Structured Graphics

Federico Mena-Quintero
Helix Code, Inc.
federico@helixcode.com
Raph Levien
Code Art Studio
raph@gimp.org

Abstract

The GNOME canvas is a generic engine for structured graphics that offers a rich imaging model, high performance rendering, and a powerful high-level API. Application programmers can use the canvas to create interactive graphics displays easily. Many GNOME applications use the canvas as their main display engine, some of them using the basic functionality provided by the canvas, and others by extending it for their particular needs. This paper describes the architecture of the canvas in detail and examines the way it is used in several GNOME applications.

1 Introduction

The GNOME canvas is a generic, high-level engine for creating structured graphics. A canvas is a window that contains a collection of graphical *items,* including lines, polygons, rectangles, ellipses, and text. The term *structured graphics* means that you can place these graphical items in the canvas and refer to them later to change their attributes. For example, a program could place a white rectangle at some specific position, and later in its execution it could change the color, position, or any other attribute of the rectangle. The canvas would then take care of all redrawing operations.

Items inside a canvas can be organized in a tree of nested *groups* which are nodes in the tree, and terminal items which are leaves in the tree. The canvas allows arbitrary affine transformations like rotation, scaling, and translation to be applied to items and groups; if a transformation is applied to a canvas group, then all of its children will be transformed accordingly. This tree organization makes it easy to create hierarchical drawings.

The GNOME canvas has an open interface that allows applications to create their own custom canvas item types. This means that the canvas can work as a generic display engine for applications. One of the following sections in this paper describes case-by-case examples of the use of the canvas in different GNOME applications.

The canvas has multiple rendering backends, one for rendering using GDK to plain X drawables[4], one for rendering using high-quality antialiasing and transparency, and one for sending the contents of the canvas to a printer.

This paper describes the architecture of the canvas and its high-quality imaging model, and presents some examples of the use of the canvas in different GNOME applications. It also describes some of the future directions for development of the canvas display engine.

2 Architecture of the Canvas

The GNOME canvas was originally based on the canvas widget in the Tk toolkit[9], which is in turn based on Joel Bartlett's *ezd* program, which provides structured graphics in a Scheme environment. The main enhancements that the GNOME canvas provides to the original Tk design are integration with the GTK+ object and signal/slot system[5], nested groups of items, generalized affine transformations, and a high-quality antialiased rendering mode. From the standpoint of the user, the canvas presents the following characteristics:

- The user is able to create graphical items like lines, boxes, ellipses, and text, place them on the canvas, and refer to them later for manipulation. The attributes of an object can be changed at any time; these include the color if the item, its line style, and its position.
- Canvas items receive events just as if they were normal X windows or other GTK+ widgets. Applications receive these events with the normal signal/slot system in GTK+. An application can then connect to the event signals of its canvas items and define the particular behavior it requires. Common actions include moving an item when the user clicks and drags it with the mouse, or highlighting an item by changing its color when the mouse pointer enters the area occupied by the item.
- Items can be grouped together in canvas item groups, and these groups can be nested within each other to form a tree structure. A canvas has a single root group which does not have a parent. Operations such as deleting or moving a group apply to all its items, thus making it easy to create hierarchical drawings. It also has performance advantages since the canvas can use recursive bounding boxes to cull out items for drawing and hit testing operations.
- Items support arbitrary affine transformations, so they can be translated, scaled, rotated, and sheared in any way. Affine transformations apply to item groups as a whole, so the children of a group will obtain the same base transformation as its parent.
- The zooming factor of the canvas can be changed at any time, and the canvas will handle all scrolling issues by itself.
- 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 wants to display.

2.1 Canvas Items and the GTK+ Object System

Canvas items are GTK+ objects derived from an abstract `GnomeCanvasItem` class, which defines several methods that all items must implement. These methods are used to perform drawing, hit testing, and updating of items when their attributes change.

Using the GTK+ object system for canvas items provides several advantages:

- No extra work is involved in wrapping them for different language bindings, since GTK+ objects and their attributes are wrapped automatically by most bindings.
- Canvas items use the usual signal/slot mechanism to emit events, making it easy for the programmer to define behavior for the items.
- One can associate arbitrary data items to canvas items, via the GTK+ dataset mechanism.

All the attributes of canvas items, like color, position, and line style, are configured and queried using the GTK+ object argument system. Canvas items may have many configurable attributes, so using the argument system allows us to minimize the number of API entry points, and also makes it easy to write language bindings for the canvas and its items — all canvas item attributes can be configured using a single function call.

2.2 Grouping of Items

Items in the canvas are organized in a tree hierarchy. Items can be groups, which are nodes in the tree, or terminal items, which are leaves in the tree. Groups can contain any number of children, which can in turn be terminal items or other groups. Items can thus be nested to an arbitrary depth inside the canvas, making it easy to create hierarchical drawings.

A canvas has a single root group. For very simple drawings or diagrams, the programmer may want to put all items directly under the root group. For more complicated, structured drawings, it will be convenient to create a hierarchical organization — a circuit editor may want to represent an adder as a group of basic logic gates, which in can in turn be groups of primitive canvas items like lines and rectangles.

The bounding box of a canvas group surrounds all of its children, so drawing and hit testing operations can be made more efficient by 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. The canvas provides functions to change the stacking order of items by raising or lowering them within their parent group's stack.

2.3 Behavior of Items

The canvas does not have any predefined behavior for items. Instead, the programmer will connect to the event signals of the different canvas items, capture events from the user, and define whatever behavior is appropriate to the application.

When an event signal is emitted for an item, it is propagated up the item hierarchy and re-emitted for its parent groups until one event handler marks the event as 'handled'. This allows the user to treat a group of items as a single meta-item; only a single signal connection is required to receive events from any of the items in a group.

Items can receive the following events: button presses and releases, pointer motion events, key presses and releases, focus in/out events, and mouse enter/leave notifications. Thus, items are very similar to normal GTK+ widgets or X windows from the programmer's point of view.

2.4 Delayed Update/Redraw Model

One of the goals of the canvas is to eliminate flicker when drawing items. Flicker is caused when an area is repainted multiple times with different colors; for example, a stack of colored rectangles would flicker if it were painted directly to the screen, one rectangle after another.

The canvas solves this problem by using a special form of double-buffering. When an area of the canvas needs to be repainted, the following actions take place:

1. The canvas creates a temporary, offscreen pixmap with the same size as the area that needs to be repainted.
2. If an item's bounding box intersects the area that needs to be repainted, the canvas asks the item to paint itself to the pixmap created in (1). The canvas thus walks the tree of items in the normal Z-order.
3. The canvas does a bitblt of the pixmap to the screen and destroys the pixmap.

The visual effect is that the whole area is painted simultaneously, eliminating all flicker.

This process actually takes place during the idle loop of the application. Repainting the canvas in the idle loop means that at that point all interaction between the application and canvas items is finished, i.e. the application literally has nothing else to do, so it is appropriate to flush all pending redraws.

2.4.1 Delayed Updates

We have not explained how the canvas actually figures out the area that needs to be repainted. Let us consider a few simple cases:

1. A solid-colored rectangle with dimensions (w, h) is translated with offsets (dx, dy) such that $|dx| < w$ and $|dy| < h$. In this case, the minimal redraw area consists of two L-shaped regions that are the symmetric difference between the old and the new rectangles (see Figure 1).

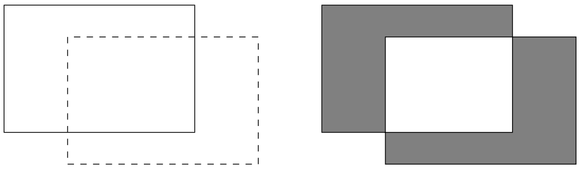

Figure 1: When a solid-colored rectangle moves, the redraw area consists of two L-shaped regions.

2. A solid-colored rectangle changes color. In this case, the whole area occupied by the rectangle needs to be redrawn.
3. A solid-colored circle changes radius. The minimal redraw area thus consists of the donut-shaped region which is, again, the symmetric difference between the old and new circles.

4. Some characters in a string of text are changed, for example, when the user is editing a label. If the string is drawn using a monospaced font, the minimal redraw area consists of the area occupied by the characters that changed. If the string is drawn using a proportional font, the redraw area will be more complicated.

One of the goals of the canvas is to compute the minimal redraw area for each operation. This is important because we wish to make the final bitblt operations as small as possible; experiments have shown that memory bandwidth tends to be a bigger problem than CPU speed.

Canvas items have two important methods, `::update()` and `::draw()`[1]. The former is reponsible for calculating the area that needs to be redrawn when the item's attributes change, and the latter gets called when the final redraw area has been computed by the canvas and an item needs to paint itself.

The full sequence of operations that starts when an item's attributes are changed and ends when the canvas paints to the screen is as follows:

1. A state change happens in a canvas item, usually from direct manipulation through the user interface. An item may thus change attributes like color or position.

2. The canvas item stores an iternal flag saying that it needs to change one of its attributes, and also stores the necessary information to change that attribute. For example, an item may store "I need to change my fill color to blue", or "my radius changed to 7.5 units".

3. The canvas item then queues an update from the canvas using the `gnome_canvas_item_request_update()` function. The canvas installs an idle handler on the GTK+ main loop.

4. The application keeps running, possibly requesting attribute changes from other items, until it finishes its work and all its interaction-related tasks, and gets back to the idle loop.

[1]Items actually have `::draw()` for canvases in GDK mode, and `::render()` for canvases in antialiased mode. However, both methods are responsible for drawing the item, so we will refer to them generically as "`::draw()`".

5. The idle handler for the canvas is run. The canvas calls the `::update()` method of each item that requested an update.

6. The `::update()` method for an item flags the item as no longer needing an update. It should recalculate the item's internal state based on the flags set in step 2, for example, by changing colors or line styles in an X graphics context (GC). This method is also responsible for recomputing the item's bounding box if the item's bounds changed. Then, it should queue a redraw from the canvas based on its new state. We will later describe how this area is represented.

7. After all items that need it have been updated, and as such they have recomputed their bounding boxes and queued the appropriate redraws, the canvas calls the `::draw()` or `::render()` method of items that need it. This method is passed a temporary pixmap in the case of a canvas in GDK mode, or an RGB pixel buffer in the case of an antialiased canvas.

8. The canvas is now fully updated and redrawn, and the application continues to run.

As we have seen, the `::update()` method is responsible for doing housekeeping work like changing GC colors and line styles, recomputing an item's bounding box, and queueing the proper redraws. This method is called from the canvas' idle handler. The reason for delaying GC updates and the like to the idle loop is that an application may change many attributes of a canvas item before getting back to the idle loop; if the item changed GCs or recomputed its bounding box for every time an attribute was changed, this could turn into a performance problem, since many such operations are expensive. Delaying all the updating work until the idle loop means that the application's interaction with items has finished, so the items know their final state at that point and can compute the most efficient way to do their respective updates.

3 The Libart Imaging Model

Up to now we have discussed the way the GNOME canvas operates internally. In this section we will describe the imaging model the canvas supports.

Libart is a library that provides a superset of the PostScript imaging model[2], and it extends it with support for antialiasing and alpha transparency. This means that the edges of graphics primitives such as Bézier paths are smoothed out to eliminate jaggies. Also, such primitives and images can be rendered and composited together using transparency information.

Libart is quite similar in design and scope to such "next-generation" imaging models as Adobe's Bravo[1], the Java 2D API[10], Adobe's Precision Graphics Markup Language (PGML)[11], and the W3C's Scalable Vector Graphics Specification (SVG)[12].

The GNOME canvas uses Libart to render its primitives when it is in antialiased mode. Also, it uses Libart's microtile arrays, described below, to represent the areas that need redrawing.

The following sections describe the main features of libart.

3.1 Vector Paths

Libart's vector paths are built from the familiar PostScript opcodes such as moveto, lineto, and curveto. Paths can be composed of multiple closed sets of segments and thus have holes in them. Paths can also cross themselves any number of times.

3.2 Sorted Vector Paths

A sorted vector path, or SVP, is a processed version of a normal vector path such that it satisfies the 'nocross' property, that is, it does not have crossing segments and retains the same winding number as the original vector path. Also, its segments are sorted so that they have monotonically-increasing y coordinates. This allows for very efficient rendering, since the segments can be traversed in scanline order.

3.3 Antialiased Rendering

Sorted vector paths can be rendered using "perfect resolution", as opposed to the common technique of rendering a higher resolution bitmap and averaging down. The SVP precomputation step is important because it allows a vector path to be rendered multiple times very quickly; canvas items compute their vector paths in the `::update()` method, convert them to SVPs, and store these so that they can re-render the SVPs quickly if needed.

3.4 Outline Stroking

This is the computation of stroke outlines for vector paths. The standard PostScript technique is to render each segment of the stroke separately, using small "miter joints" added at each corner. However, this technique is not ideal for antialiased rendering because the resulting number of polygons is large, and the adjoining polygons can produce seams and other artifacts.

Libart creates stroke outlines by computing inner and outer contours around the original vector path, and then performing a boolean union on them. This union operation cleans up any intersections or overlaps of the stroke. The result is a pure vector path that can be efficiently rendered with the usual algorithm. This is faster and visually more precise than rendering each segment of the stroke separately.

The stroking algorithm supports the same line join and cap options as PostScript.

3.5 Vector Path Operations

Libart can perform intersection (clipping), union, difference, and symmetric difference of vector paths. The latter is especially important for exact computation of redraw areas in the canvas.

3.6 Raster Images

Libart supports affine transformations for raster images, so they can be scaled, rotated, and sheared in any way. It also supports full alpha transparency for images and special gamma correction for opacity.

3.7 Microtile Arrays

An important requirement for the canvas is to have an efficient representation of the area that needs to be redrawn. This area can be disjoint and potentially very complex, since items can be scattered across the canvas area and one would wish to avoid painting a single large bounding box for all of them.

Microtile arrays are a simple data structure for representing 2D regions, suited to representing redraw areas.

The array divides the area into a grid aligned on 32-pixel boundaries. Within each grid square is a bounding rectangle, the microtile. Since all coordinates in the microtile are in the range $[0, 32]$, 8 bits are more than sufficient for each coordinate. Since each microtile requires four coordinates to represent its bounding rectangle, each microtile can be conveniently represented with a 32-bit value.

Figure 2 presents the microtile representation of a complex area. The polygon defines the covered area. The shaded region represents the individual microtiles; each microtile is the bounding box within a grid square that surrounds the covered area. For the final redrawing operation, adjacent microtiles are combined together into bigger rectangles, shown as thicker outlines in the diagram. Each of these rectangles is then calculated and drawn to the screen.

Microtile arrays have many advantages which make them ideal for the canvas widget. First, the data structure is compact; a microtile array for a 640×480 window requires only 1200 bytes. Second, it can be manipulated very quickly; the sample polygon in Figure 2 is calculated in about 1.2 ms on a 233 MHz PII. Third, the resulting microtile array is easily decomposed into rectangles.

Rectangle decomposition has several desirable properties, including a bounded number of rectangles — no more than 300 for that 640×480 image, no matter the complexity of the area. Also, rectangles tend to align on 32-pixel boundaries, which can speed things down the rendering pipeline.

In the context of the GNOME canvas, items can queue their redraw areas in any of three ways: they can specify an explicit microtile array, in which case it is added to the canvas' current redraw area; they can request that a rectangular area be redrawn, and so the canvas converts that rectangle to a microtile array; or they can specify a sorted vector path as the redraw area, which is again converted to a microtile array.

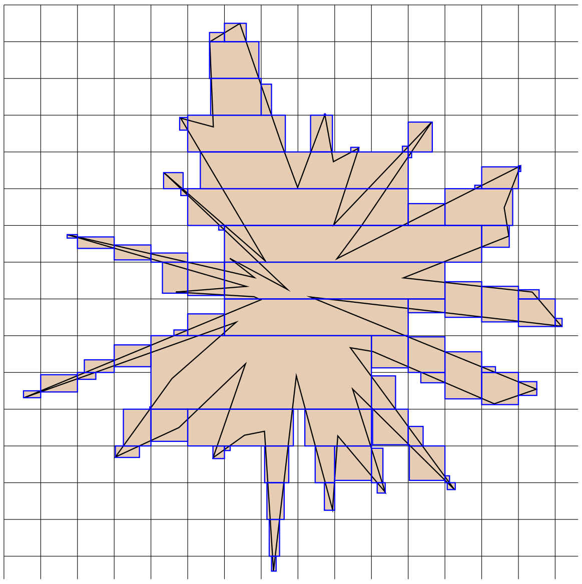

Figure 2: Microtile array that represents the area occupied by a complex polygon. Each little bounding box, or microtile, fits within a grid square. Microtiles are later coalesced into bigger rectangles suitable for redrawing.

3.8 Miscellaneous Utilities

Libart provides miscellaneous functions to handle affine transformations, points, and rectangles. Affine transformations can be applied to vector paths or raster images. Rectangles are used to represent bounding boxes and other things that are useful to the canvas and applications in general.

3.9 The Updating and Rendering Pipeline

Figure 3 presents an illustration of the updating and rendering pipeline of the canvas. The steps are as follows:

1. The first step is to cause a state change in a canvas item, usually from direct manipulation through the user interface.

2. Identify the deltas, or the parts of the display that have actually changed. This is done in the `::update()` method of canvas items.

3. Represent the deltas as a microtile array. Each microtile is a small rectangle that needs updating.

4. The microtile array is decomposed into bigger rectangles for more efficient redrawing.

5. Fifth and sixth, repeated for each rectangle from (4), canvas items are rendered in their normal stacking order, culling them against the bounding boxes defined in (4), and are displayed in the canvas window.

4 Applications that Use the GNOME Canvas

This section describes how different GNOME applications use the canvas as their display engine.

4.1 Gnumeric

Gnumeric is the GNOME spreadsheet program[7]. It uses the canvas as its main display engine to allow for easy event handling, extensibility through components, and flicker-free display.

Gnumeric defines several custom canvas item types:

- An `ItemGrid` item which takes care of displaying all the cells in the spreadsheet. This draws the actual grid and the cell contents, with support for different fonts and colors.

- An `ItemCursor` item that takes care of displaying the specialized selection and active cell combo 'cursor', as well as its decorations. The selection has a thick outline which the user can drag to move cells around, and it also has a little rectangle that can be used for the auto-fill function.

- An `ItemBar` item that displays row and column headings for the spreadsheet. The user can drag the edges of the "buttons" that represent rows and columns to resize these. The user can also click and drag on the buttons themselves to select whole rows or columns in the spreadsheet.

In addition, Gnumeric uses some of the primitive canvas item types such as rectangles, ellipses, and lines to display miscellaneous elements of the user interface.

4.2 Gnome-PIM

Gnome-PIM is the GNOME personal information manager, which consists of a calendar and a contact manager or addressbook.

The calendar program needs to present many interactive graphics displays, such as monthly calendars with captions for appointments, yearly calendars with marked busy days, and other views for weeks and days.

The contact manager program needs to display a familiar representation for "business cards", or the data that describes a personal contact.

Instead of drawing all of these displays by hand, Gnome-PIM uses the canvas to create these displays out of primitive canvas items such as lines and rectangles. This allows the application to invest more effort in event handling to give the best possible experience to the user, while leaving all the tricky display issues to the canvas.

4.3 Evolution

Evolution is the next-generation mail and groupware program for GNOME. The mailer requires many complex displays such as a hierarchical view of mail folders with previews of the first few lines of mail messages; the rest of the PIM-related modules require calendar and business card displays.

Evolution defines an `ETable` canvas item that implements a model/view/controller abstraction for the display of tabular data. Custom cell renderers can be plugged into this item, turning it into a general-purpose grid display. The canvas allows this complex item to do flicker-free display easily.

Some of the information displays used in Evolution

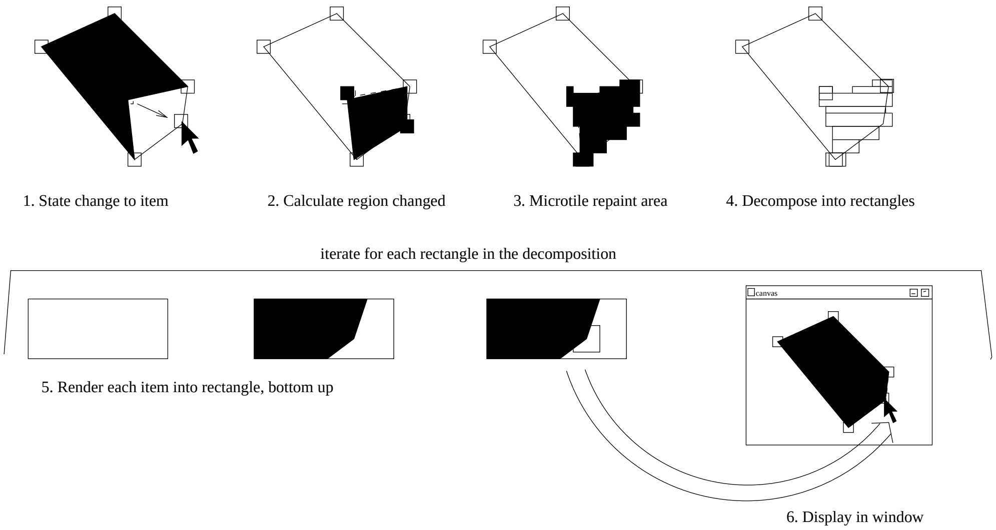

Figure 3: The updating and rendering process

are very complex, as they need to present the user's personal information in a convenient way. The canvas allows Evolution to concentrate on presentation and user interface issues rather than mundane tasks such as redrawing and event handling.

4.4 Eye of Gnome

The Eye of Gnome program is the GNOME image viewing and cataloging program. It uses the canvas for its main image display, and it defines a custom canvas item that can do extremely fast scaling of images suitable for an image viewer.

EOG also defines a model/view/controller abstraction for "wrapped lists", and in turn implements an icon or thumbnail view for large sets of images. It uses special techniques so that only the icons and captions that fit in the canvas window actually exist as canvas items; these are created and destroyed on the fly as the icon list is scrolled and updated.

The canvas allows for easy event handling, and with the delayed update model, also allows for on-the-fly loading and generation of thumbnails.

4.5 Gnome-print

Gnome-print is the GNOME printing framework. It uses the Libart imaging model so that applications can enjoy the same rich imaging model as the canvas' for printing.

In addition, there is a special printer driver for Gnome-print that takes in all the PostScript-like commands and creates canvas primitives for them instead of sending them directly to a printer device. Thus, whatever the application prints is transformed to Bézier paths that appear as items in a canvas. This can be used as a simple "print preview" widget by applications. The canvas allows automatic zooming and scrolling, so applications do not have to be modified at all to support a high-quality print preview.

4.6 Illustration programs

Sodipodi is a powerful illustration program that uses the antialiased canvas for its display. It uses many Libart operations to do computations on Bézier paths.

Gill is a testbed for illustration-related tasks. It parses SVG files and creates the corresponding Bézier, image, text items in the canvas. One of its goals is to support external manipulation of objects via the DOM.

4.7 BEAST

BEAST is a music program that uses BSE, the Bedeviled Sound Engine. It uses the canvas to display filter pipeline graphs and envelope functions for waveforms.

4.8 Gnoghurt

Gnoghurt is a toy program to create video filter pipelines. It uses the canvas to let the user edit these pipelines in an easy way. Gnoghurt provides fruit at the bottom, and must be stirred before eating.

5 Future Work

The canvas has some room for improvement. Here are some of the possible directions in which it may be extended in the future:

- The W3C's Scalable Vector Graphics Specification, or SVG, supports fully hierarchical clipping and opacity adjustment. For example, a whole group of items may be clipped by the result of performing boolean operations on another group of items. Also, the alpha transparency value for a whole group can be changed simultaneously.

  The canvas could be extended to support these operations. Groups are already rendered recursively, so controlling the alpha value for a group would be a matter passing the parent's alpha value to its children The result would later be composited onto the group's parent's buffer, and so on recursively. Clipping with the result of boolean operations on other groups is more complex, but can be represented in a hierarchical fashion similar to the canvas' current organization.

- Right now the Bonobo component system[3] can be used to proxy canvas items to remote components. This allows for embedding of non-rectangular components in documents. However, this is not as efficient as it could be.

  The delayed update/redraw model of the GNOME canvas assumes synchronous updating and redrawing of individual canvas items, so this is not very efficient for a distributed setting where multiple components may be running on different machines or even on a single multiprocessing machine. The canvas could be extended to support fully asynchronous updates and redraws, asking items to draw themselves in parallel to temporary buffers which would be composited on the fly as they arrive to the parent canvas.

  This would allow components running on different processors to draw themselves in parallel and notify the toplevel canvas when they are finished; the canvas would then composite their buffers to form the final result that would be displayed on the screen.

6 Acknowledgments

The GNOME canvas is a collaborative effort. The original version of the canvas was based on the Tk canvas widget. Federico Mena adapted it to the GTK+ object system and extended it with hierarchical groups. Raph Levien wrote the Libart engine and extended the original, GDK-only canvas to support it.

Many people have contributed with ideas and bug fixes to the canvas. The Gnumeric hackers provided excellent bug reports, and coped with our delays in fixing them. Tim Janik pointed out the most bizarre bugs in the antialiased canvas. Owen Taylor provided the scrolling backend for the canvas and was always knowledgeable and helpful with the intricacies of the X window system.

Red Hat, Inc. funded a large part of the development of the canvas, and provided a fun work environment and knowledgeable people. Helix Code, Inc. funded mainteinance work of the canvas.

7 Availability

The GNOME canvas is part of the standard `gnome-libs` package, which consists of the core libraries in GNOME. It can be obtained from `ftp.gnome.org` or from the GNOME CVS repository at `cvs.gnome.org`.

Documentation for the canvas is available at `http://developer.gnome.org`.

References

[1] Adobe Systems, Inc., Bravo Technology Announcement.

[2] Adobe Systems, Inc., *PostScript Language Reference Manual*, third edition, Addison-Wesley, 1999.

[3] The Bonobo Component Framework, `http://developer.gnome.org/arch/-component/bonobo.html`, `http://developer.gnome.org/doc/-guides/corba/book1.html`.

[4] The GIMP Drawing Kit (GDK), `http://www.gtk.org`, `http://developer.gnome.org/arch/gtk/-gdk.html`.

[5] The GIMP Toolkit (GTK+), `http://www.gtk.org`, `http://developer.gnome.org/arch/gtk`.

[6] GNU Network Object Model Environment (GNOME), `http://www.gnome.org`.

[7] Miguel de Icaza, *The Gnumeric Spreadsheet: a Test-Bed for Component Programming*, Proceedings of Linux Expo 1999, `http://www.gnome.org/gnumeric`.

[8] Raph Levien, *GtkCaanvas and the Next Generation of User Interfaces*, Proceedings of Linux Expo 1999.

[9] John Ousterhout, *Tcl and the Tk Toolkit*, Addiwon-Wesley, 1994.

[10] Sun Microsystems, Java 2D API, `http://www.javasoft.com/products/-java-media/2D/index.html`.

[11] World Wide Web Consortium, Precision Graphics Markup Language Specification (PGML), `http://www.w3.org/TR/1998/NOTE-PGML`.

[12] World Wide Web Consortium, Scalable Vector Graphics Specification (SVG), `http://www.w3.org/Graphics/SVG`.