The

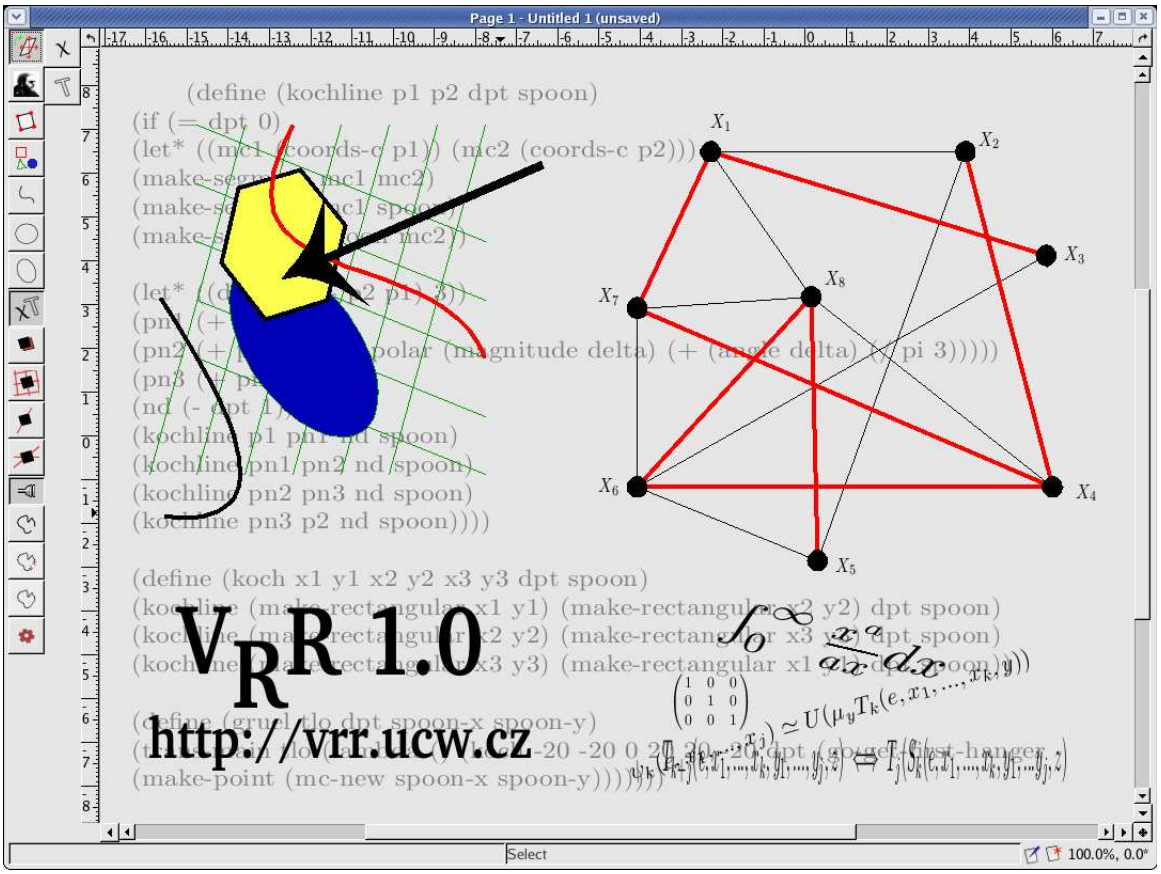
VRR

User's Manual

Table of Contents

1 Introduction

VRR (a Vector-based gRaphic editoR) is an application designed especially for creating illustrations of mathematical articles.



Picture 1: The logo.

VRR has a simple but powerful operation set: creating, manipulating and transforming basic graphic primitives, which are points, segments, rational Bézier curves, elliptic arcs etc. All objects can be determined not only by absolute coordinates, but also by geometric dependencies on other objects – intersections, significant points, other curves etc. When an object is changed, the dependent objects are recalculated automatically. This enables you to modify the image easily without breaking the lines visually tied together.

VRR has a sophisticated system for working with text. In addition to common text objects, it allows you to create text objects typeset by TEX and make them part of your image. You can create almost anything from a short math expression to several paragraphs of a text.

The editor is able to import from and export to files in common data formats (export to PDF, PS, EPS and SVG, import from IPE5 and SVG).

VRR is a freely available open source program, which runs on Linux and uses a graphical user interface based on the X Window System and GTK. The authors suppose that the time spent by a user exploring VRR's interface will bring him much joy later when he is able to use all of its powerful advanced features.

The editor can be also easily extended by a plugin or by a program written in its built-in scripting language, Scheme.

If you have any suggestions or bug reports, you can contact the VRR Team on vrr@ucw.cz. For new release versions of VRR, documentation and many related articles, see the VRR developers' page – http://vrr.ucw.cz/.

2 Installation Instructions

2.1 Installation Requirements

Hardware requirements

VRR has no specific hardware requirements except for

- a computer able to run a decent *NIX operating system (Linux presumably) and X Window environment (not required for the commandline-only interface),
- graphical display (not required for the commandline-only interface),
- a mouse with at least two buttons, three buttons are even better (again, not for the commandline-only interface).

To prevent difficulties with long reaction times while running VRR, it is also recommended to have a system fast enough to perform sophisticated geometric computations.

Software requirements

VRR was developed under Linux operating system, but if all prerequisities are fulfilled, it can run under any of *NIX-like systems.

Before the installation of VRR, make sure your system fulfils these prerequisities:

- GTK+ 2.6.0 or higher
- Guile library 1.6 or higher
- LibKPathSea
- FontConfig 2.3.1 or higher
- pdfTEX
- ZLIB
- LibXML 2.0 or higher

This is enough only when installing VRR from a binary package. To compile VRR on your own you need also these:

- GNU make
- gcc 3.0 or higher
- GNU awk
- Perl

In most cases you will need the development packages (libXXX-dev) of the previous libraries to compile VRR.

The program is also able to use the libpaper library for paper sizes management and Cairo library for advanced rendering, but they are not mandatory for successful compilation.

Optional: The documentation books are distributed along with VRR. No compilation of the documentation is needed; however, if for any reason you want to build the documentation yourself, you will need the GNU Texinfo (version at least 4.7) and various utilities, like texi2html. For instructions how to build the documentation, see the Programmer's Manual.

2.2 Compilation

There are two ways to install VRR. Either you can download the source files and compile VRR, or you can download and install the x86 binary as a Debian package. We will now describe the compilation.

Download the gzip or bzip2 archive from the website http://vrr.ucw.cz/. Unpack the archive and change directory to its root directory. Then type `make config` to configure your installation automatically or run `build/configure` to configure your installation manually. The `build/configure` script accepts many options as other configure scripts do; for help about these options, run `build/configure --help`.

If you have the Cairo library installed and you want to use it instead of GDK for image rendering, then you can enable it manually during configuration.

When the configuration has finished, type `make final` to compile the program.

Optional: If you do not want to install the program, you can still compile and run it. Instead of `make final`, run `make local` only to compile the program. To run the program, run `bin/vrr` or `bin/guile-vrr` in the '`run`' directory.

2.3 Installation

First make sure you have the write permission to the destination directories. In most cases you will need to login as root. Then type `make install`. The two main binaries: the GUI-version `vrr` and non-GUI Scheme console `guile-vrr` will be installed. The other installed binaries are tools:

- `pfb2pfa` for converting binary Type1 font into ASCII Type1 font
- `pfa2pfb` for reverse conversion
- `tt2t42` for converting TrueType font into Type42 PostScript font

Together with binaries, libraries and data files, also documentation files and examples are installed. You can choose the installation path by yourself by `build/configure`, the default paths are usually '`/usr/share/vrr/doc`' or '`/usr/local/share/vrr/doc`'.

Removing the binaries

After the installation, you can remove the generated binaries from the source tree by typing `make clean` or you can do an even better cleanup by `make distclean`.

3 Tutorial

In this chapter, you will learn step by step how to create graphic objects. We start with the simplest objects and guide you through all possible ways of creating, manipulating and transforming objects to make sophisticated images.

Basically, VRR works with documents. Each document is stored in one file and may contain several pages, with every page considered an independent image. To start editing a new image, first create a new document by clicking the  icon or use the 'File/New' (*Ctrl + N*) command. A new file with one page will be created and a window displaying the contents of the page (the View window) appears. Now you can start creating new graphic objects in the page.

3.1 The first simple graphic objects

3.1.1 Creating graphic objects – overview

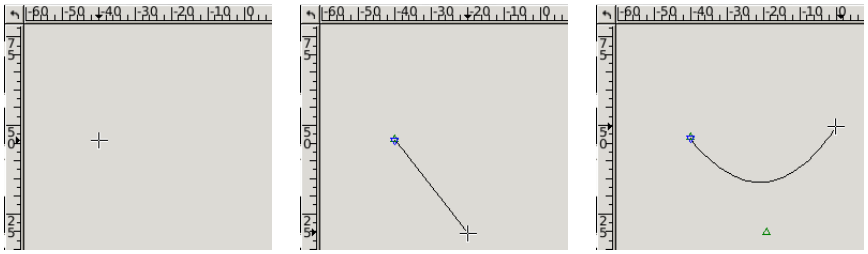
In the toolbars on the left of the View, you can see many icons. Some of them enable you to create new graphic objects. The icons are grouped in several categories; by clicking the category icon on the left toolbar, the icons of that category are expanded into the right toolbar.

These icons are:

-  **Points and decorations**
  -  a point
  -  adecoration point
  -  an arrow
  -  an intersection of two curves
  -  an n-gon defined by the number of apices, center and one apex
-  **Bézier curves**
  -  a segment
  -  a quadratic rational Bézier curve
  -  a cubic rational Bézier curve
-  **Circles and circular arcs**
  -  a circular arc defined by three points
  -  a circular arc defined by the center point and radius
  -  a circle defined by three points
  -  a circle defined by the center point and a point
  -  a circle defined by the center point and radius
-  **Ellipses, elliptic arcs**
  -  the smallest ellipse defined by three points
  -  an ellipse defined by three points, rotation, eccentricity
  -  an ellipse defined by two foci and a point
  -  an ellipse defined by the center point, a point, eccentricity
  -  an ellipse defined by the center point and radii
  -  an elliptic arc defined by the center point and radii
-  **Texts**
  -  a TEX text
  -  a text

Choose an icon in the right toolbar and click it. Now you are ready to create your first graphic object. Each object is determined by a certain number of points, for example, a segment is determined by two points: its start point and end point. Read Section 5.3.4 [GO Creating Modes], page 41 for further details. Once you have clicked the appropriate icon, you must specify the points to determine the desired position of the object. This is done by clicking the drawing area with the left mouse button.

When choosing the points, you can see the object being created and getting its shape according to the chosen points and the current position of the mouse cursor. In the status bar of the View, there are hints that remind you what point is to be chosen at the time. Once all points are chosen, the object is finished and you can either switch to another object type by clicking some other icon, or you can start creating an object of the same type immediately.



Picture 2: Creating a new Bézier curve

When choosing the points, you may want to return one step back. That can be done by pressing `BackSpace` – your last click gets forgotten and you can continue as if it had never been done. By pressing `BackSpace` several times, you can return several steps back up to the beginning of the currently created object. Or, by pressing `Esc` you cancel the creation and delete the object at once.

When you create graphic objects, you can see small green and blue triangles emerge in the positions where you clicked. These are not part of your image, they just mark the significant points. Their meaning is described in more detail in Section 4.1 [Anchors and hangers], page 27.

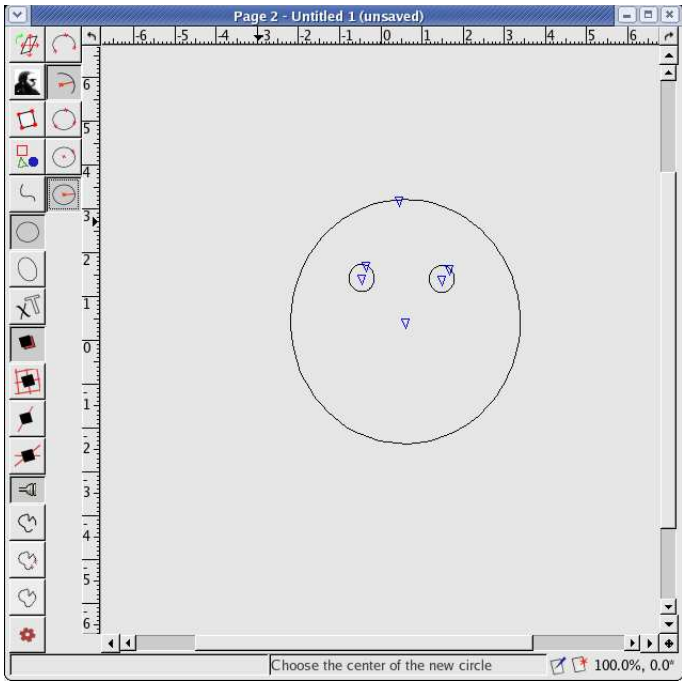
Note: Some objects available need to be determined by data different from a point position. For example, when creating an n-gon, you are asked for the number of its apices. When creating a text object, an editor opens for you to specify the source text of the object (see Section 5.1.6 [The Text Editor], page 35).

3.1.2 Creating graphic objects – the cat example

In the following example, we will create the following picture of a cat:

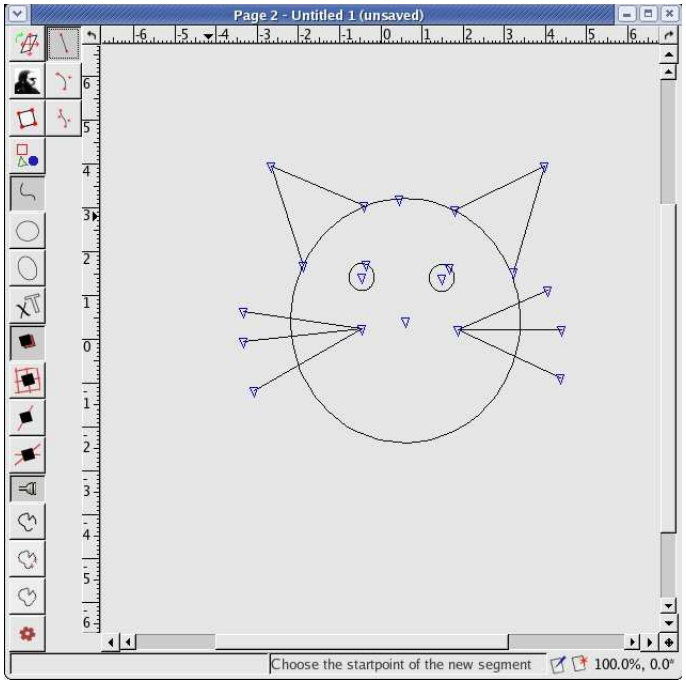
Picture 3: The cat example.

We start with circles. Create three circles by center and point to form the head and eyes of a cat like in this picture:



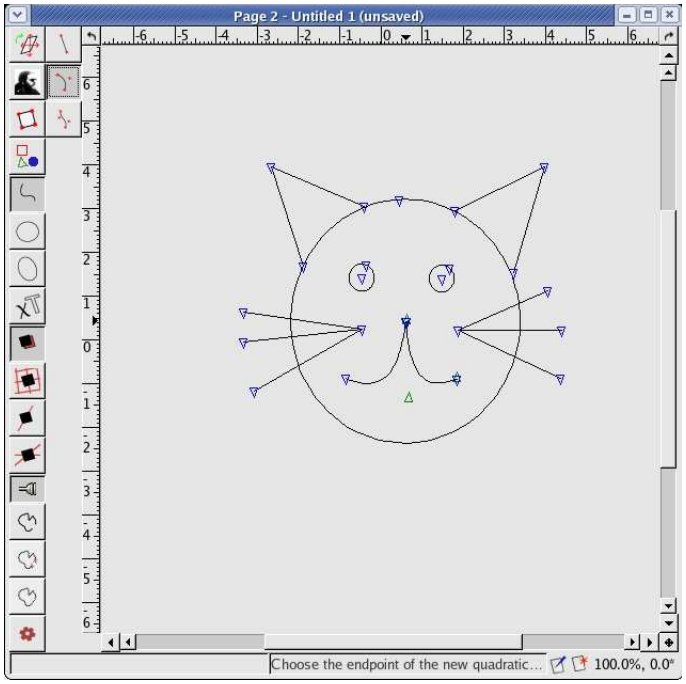
Picture 4: The head and eyes of a cat.

Now continue with segments . Draw ten lines – the ears and whiskers. The picture should now look like the following picture:



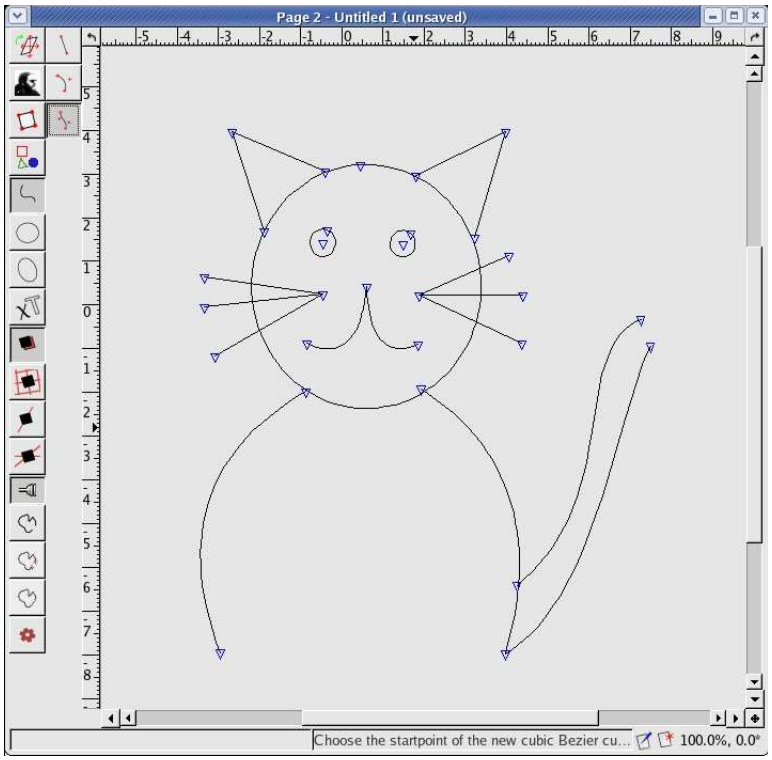
Picture 5: The ears and whiskers.

Create the cat's mouth using quadratic Bézier curves :



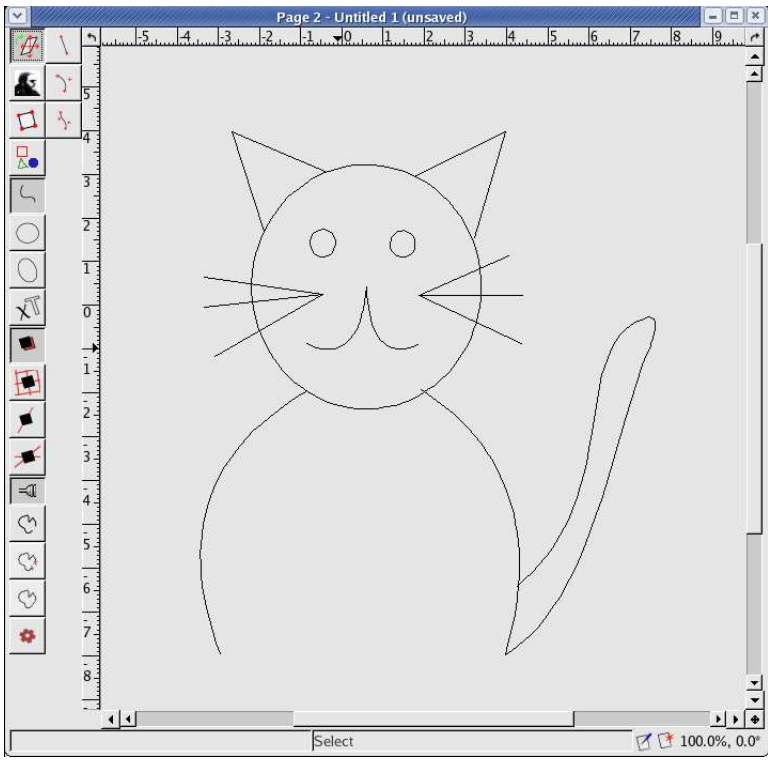
Picture 6: Bézier curves to form the cat's mouth.

Now draw cubic Bézier curves  (the cat's tail):



Picture 7: The cat's tail.

At last finish the picture with a quadratic Bézier curve. The resulting picture should look like this:



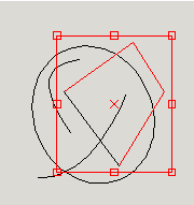
Picture 8: The resulting picture.

3.2 Selection

To manipulate with the created objects, you need to specify which objects to work with. Most tools work with the *selection*.

The simplest way to select an object is by clicking it in a View. First choose the Select/Transform tool . By *Shift* + click you add an object to selection and by *Ctrl* + click you remove it from selection. Clicking an object without any of *Shift* or *Ctrl* pressed clears the selection and makes it the only selected object in the page. The keyboard shortcuts work similarly for rectangular selection. Instead of plain clicking, press the left mouse button, by dragging it define a rectangular region, and release. This modifies the selection with all objects inside the region. In addition, the "Edit/Select all" command (*Ctrl* + *A*) selects all objects in the page and the "Edit/Clear selection" command (*Shift* + *Ctrl* + *A*) clears the selection. You can also clear the selection by clicking the drawing area far enough form any object.

You can see a red rectangle bounding all selected objects. Also, the selected objects are drawn in red instead of their real color. This helps you to recognize which objects are selected and which are not.



Picture 9: The red selection bounding box.

The selection is local for pages, which means that selection changes in one page do not affect selection in other pages at all. (This is true not only for selection, but for all editing actions of page contents. Undo history is local for pages, too.)

Another way to select objects is to do that in the Universe browser window (see Section 5.1.3 [Universe Browser], page 32).

3.3 Basic actions

The right mouse button in the View opens a pop-up menu. All editing actions available in the View can also be found in the menu. We stress the most important of them:

Delete, Cut, Copy, Paste

All these basic actions work with the selected objects and have the usual keyboard shortcuts: *Delete*, *Ctrl* + *X*, *Ctrl* + *C*, *Ctrl* + *V*, respectively. After paste, the current selection is modified to contain exactly the pasted objects.

Undo and redo

Each page has its own undo history which is independent of undo histories of other pages. Therefore, you can undo and redo actions performed in a page without undoing or redoing any action in other pages, regardless of their global historical order.

Optional: There is also a global undo history which keeps track of actions not belonging to any page contents.

Save, load, export and import

All documents created in VRR can be saved or exported to a file in one of the several supported graphic formats. To save a document, use the "File/Save" menu item. The default format is the VRR native format – a lisp-like text file described in the VRR Programmer's Manual. The file can be loaded again using the "File/Open" menu item.

The export menu items can be found in the "Export" menu category. The available formats are:

- **PDF** – export the whole document, one page per page
- **PS** – export the whole document, one page per page
- **EPS** – export the one specified page only
- **SVG** – exports the one specified page only

VRR also allows importing pages from SVG and IPE5 native format. Choose a page into which you want the image to be imported. Then choose "Page/Import SVG" or "Page/Import IPE5" to import the image.

In SVG import, VRR supports basic graphic objects and their properties. It does not support groups, filters, gradients, triggers, aliases and cascade styles. IPE5 supports all features except for splines, patterns, colors and fills (we plan to improve it in future releases).

3.4 Transformations of graphic objects

All the transformation tools of VRR allow you to transform and view the transformation changes continuously. If the process is too slow, then only some mouse cursor position changes are processed (but the button release is always processed so that the dragged object gets exactly on the position where you dropped it). In case that a transformation step fails, the objects stay at their last well-defined positions and wait for another successful transformation step.

3.4.1 Transforming using the Select/Transform tool

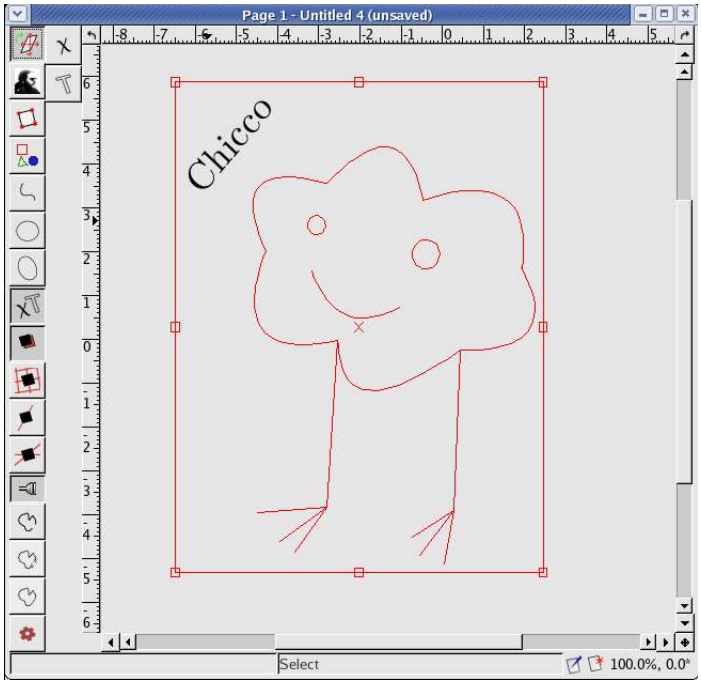
Now we will show how to perform affine transformations (like move, rotate, skew) on geometric objects. Switch to the Select/Transform mode by clicking the  icon. Select the objects you want to transform (see Section 3.2 [Selection], page 10). On the red bounding box of selection, there are small squares of various colors. These are the transformation gadgets, each gadget stands for a certain transformation. While holding the ***Shift*** key, you can see different transformation gadgets.

Move

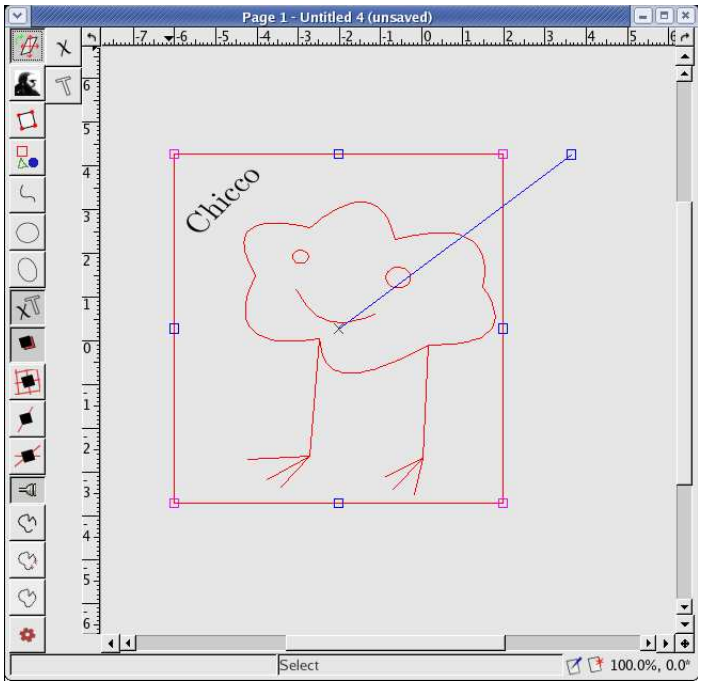
By dragging the red cross in the middle of the bounding box or by dragging the box boundary (far enough from a gadget), you move the selected objects.

Resize

On the bounding box, there are eight red gadgets – in each corner and in the middle of the sides of the rectangle. The gadgets in the middle of each horizontal side resize the objects vertically and conversely. The horizontal size remains unchanged, thus this action modifies the aspect ratio. To scale equally in both directions (and keep the aspect ratio unchanged), use the corner gadgets. The red cross is the fixed point of the resize transformations.



Picture 10: Transformation gadgets in the normal mode.



Picture 11: Transformation gadgets in the *Shift* mode.

Rotate

While holding the *Shift* key pressed, the corner gadgets become magenta and then by dragging them, you rotate the selected objects. Here, the red cross is the fixed point (the center of the rotation), too.

Skew

VRR also allows you to skew selected objects. Like for rotation, during the whole operation you have to keep the *Shift* key pressed. The blue segment which appears while pressing the *Shift* key represents the skew axis. Only the direction (and not the length of the segment) is important; the line coming through the segment contains all the fixed points of the transformation. Now drag the blue gadgets (in the middle of the bounding box sides) to skew according to the axis. Notice that dragging the gadget in the direction perpendicular to the axis has no effect, only the parallel movement causes the skew.

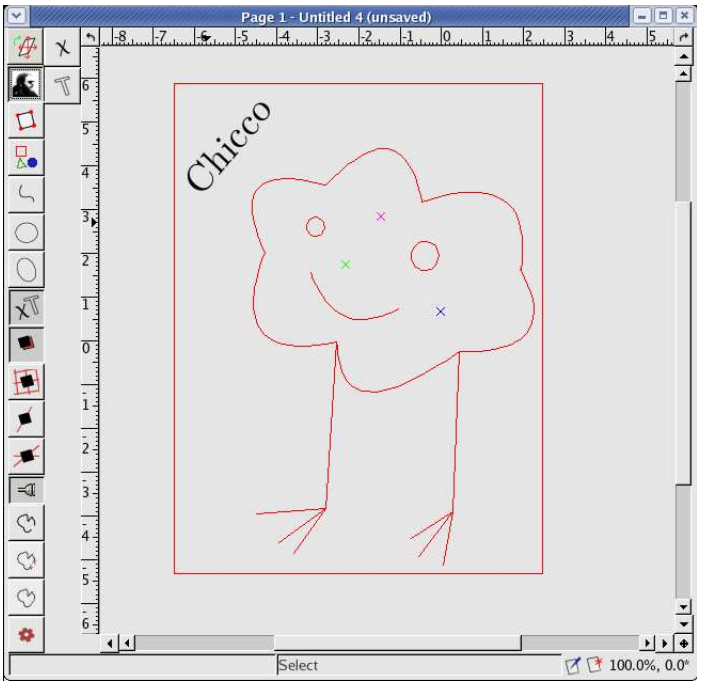
Edit the fixed points

While holding the *Shift* key, you can also change the position of the red cross and the skew axis point by dragging them.

3.4.2 Santiago's transform tool

All affine transformations can be done using the special Santiago's transform mode as well. First, select the objects which should be transformed (see Section 3.2 [Selection], page 10). Then press the  icon to switch to this mode.

Having the *Shift* key pressed, click to position the three transformation crosses. The first click places the first cross, the second one places the second cross etc. By clicking an already positioned first or third cross, you remove it. By clicking the second cross, you toggle it between the states blue, red, removed. If a cross with lower number is removed, the crosses with higher number are disabled (gray).



Picture 12: The three transformation crosses of Santiago's transform mode.

Now drag a cross to transform. For all crosses, the transformation is computed in such a way that the point from the original cross position becomes the point at the new cross position.

The first (magenta) cross is the gadget for move.

The second (blue/red) cross is the gadget for resize/rotate, which are all possible linear transformations (may be combined together). The first cross is the fixed point.

The third (green) cross does the skew. The former crosses (and the line connecting them) are the fixed points. This enables you to do all affine transformations.

Note: The transformation in the Santiago's tool is computed according to the relative positions of the three crosses among one another. It does in no way depend on their absolute position in the image (or with regard to the selection bounding box), which might seem somewhat surprising at first.

3.4.3 Predefined basic transformations

There are also some predefined transformations. They can be found in the Edit/Transform category of the View pop-up menu. They are: flip vertically, flip horizontally, rotate by 90 degrees, rotate by 180 degrees, rotate by 270 degrees; they work with the selection, too.

3.5 Modifying the properties

3.5.1 Modifying the properties – Introduction

In the previous parts we have learned how to create, manipulate, and transform a graphic object. There are also other ways how to alter the object's appearance. Select the object (see Section 3.2 [Selection], page 10 to find out how) and open the Property window by choosing either the "Edit/Properties" or the "Windows/Property window" menu item. The two ways differ slightly, as described in Section 5.1.5 [The Property Window], page 34. Basically, the Property window opened in the latter way reacts to selection changes and updates the displayed objects accordingly. Also, you can edit properties of multiple objects in it.

The property window contains the list of all properties of the selected object(s). You can change their values here; however, the attempt to change some of them may fail for various reasons, mainly the geometric dependencies. If you try to change the coordinates of the center point of a circle determined by three points on its perimeter, it fails. In that case, the value remains unchanged.

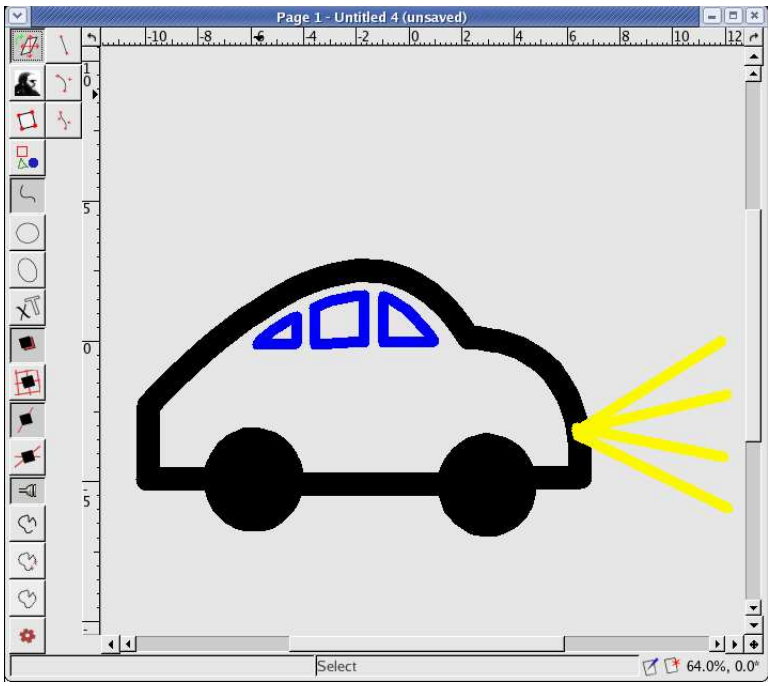
Some values can be edited directly by putting down the new value or using the arrows in spin buttons (edit boxes with arrows that changing the value by certain predefined steps). Some values are to be chosen from the list and for some property types, an editor will be opened (text, color, . . .).

The "invisible" property allows you to make any object invisible on the drawing area. But physically, it is still in the object tree (see Section 5.1.3 [Universe Browser], page 32) and can play part in geometric dependencies.

You can also create, edit, and delete your own properties; that can be useful for making some notes or in script processing. To add a property, press the "Add . . . " button and fill in the name (the key), type, subtype and value of this property. Note that the property key must be unique. To delete some properties, select them by clicking their names in the Property window (or unselect by clicking once more). Then press the "Delete" button. Note that not all properties can be deleted; some of them are important for geometric and graphic features of the object. But you can always delete the properties you created yourself. Even these property actions can be undone and redone using the local undo/redo.

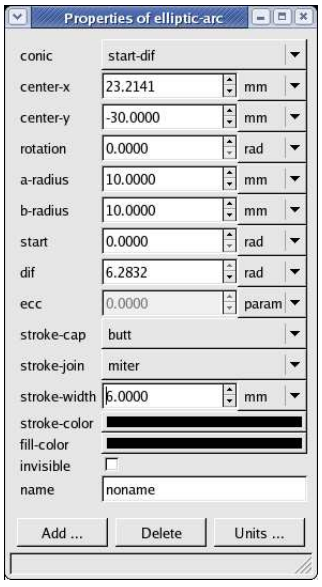
3.5.2 Modifying the properties – The car example

We will create a simple picture and try to modify the properties of some graphical objects:



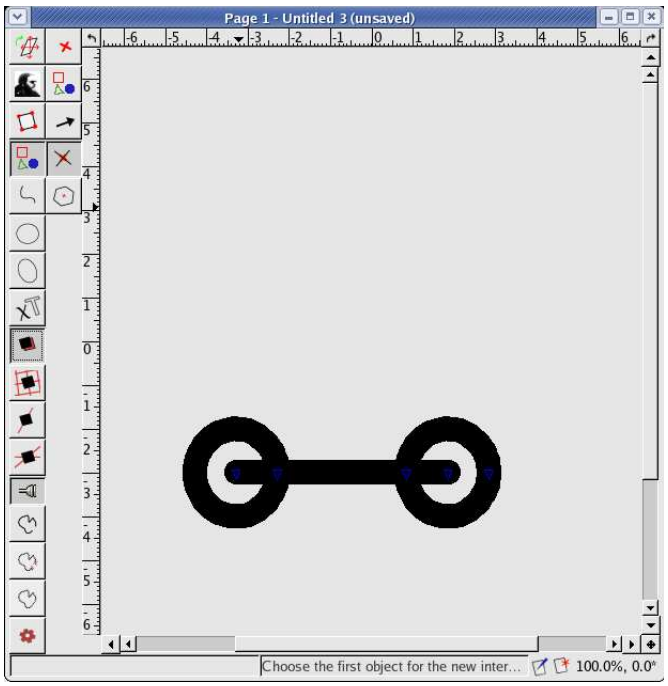
Picture 13: The picture of a car drawn in thick colorful lines.

First, draw a circle (defined by the center and the radius) and in the property editor change the stroke-cap to round and property "stroke-width" to 9 mm. The values you set are saved and used again for newly created objects. Create another circle and it will be drawn with stroke width 9 mm.



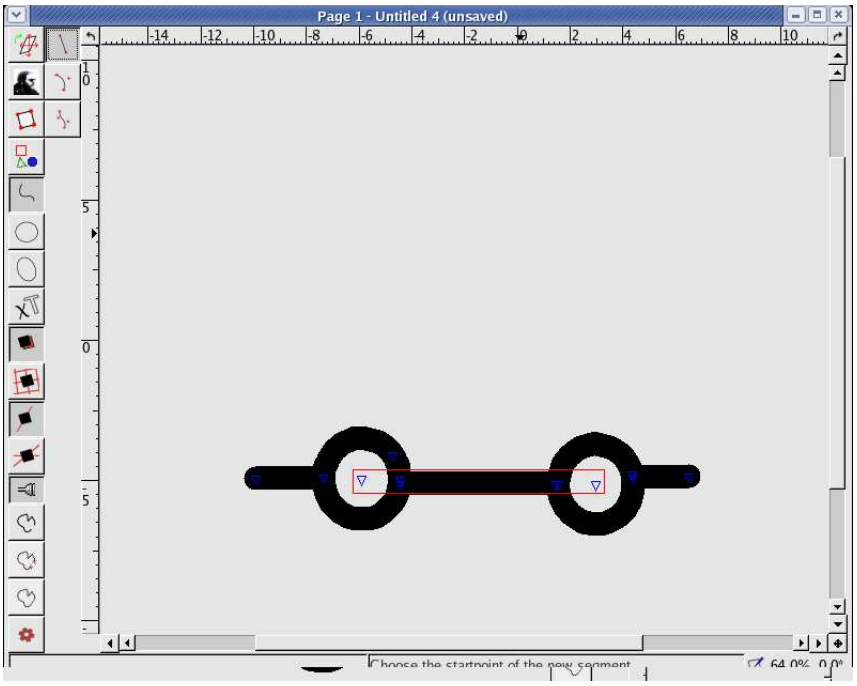
Picture 14: The Property editor window.

Now change the "center-y" coordinates and the "radius" properties in both circles to the same value (the circles will be aligned horizontally and have the same size). Draw a segment with end points in the centers of the two circles. Now create two intersections of the segment and both circles (using the icon).



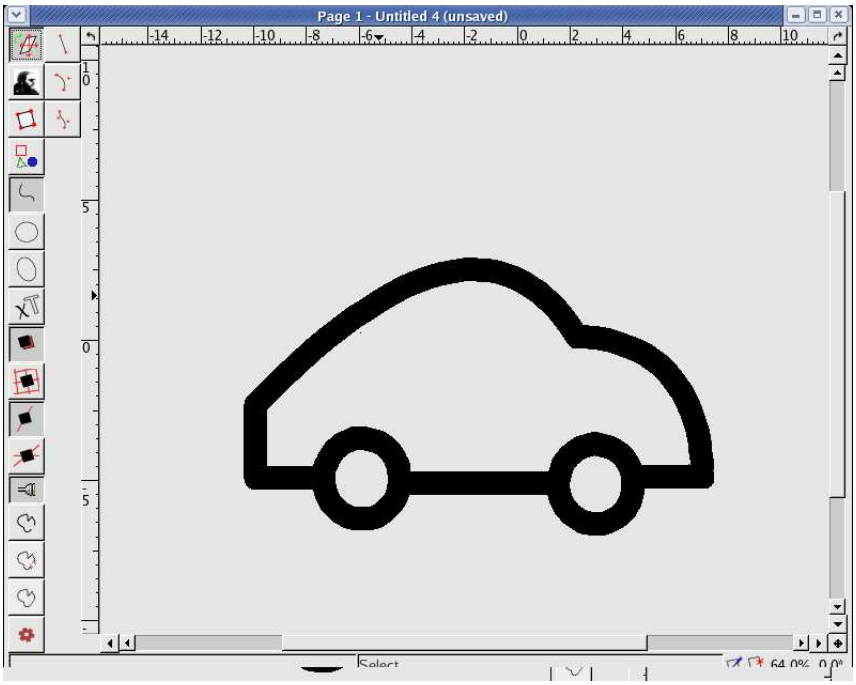
Picture 15: A black line connecting the center points.

Select the segment and make it invisible (set the value of the "invisible" property to true). Now let these two intersections make a segment. Draw another two lines (car chassis). The picture should look like this:



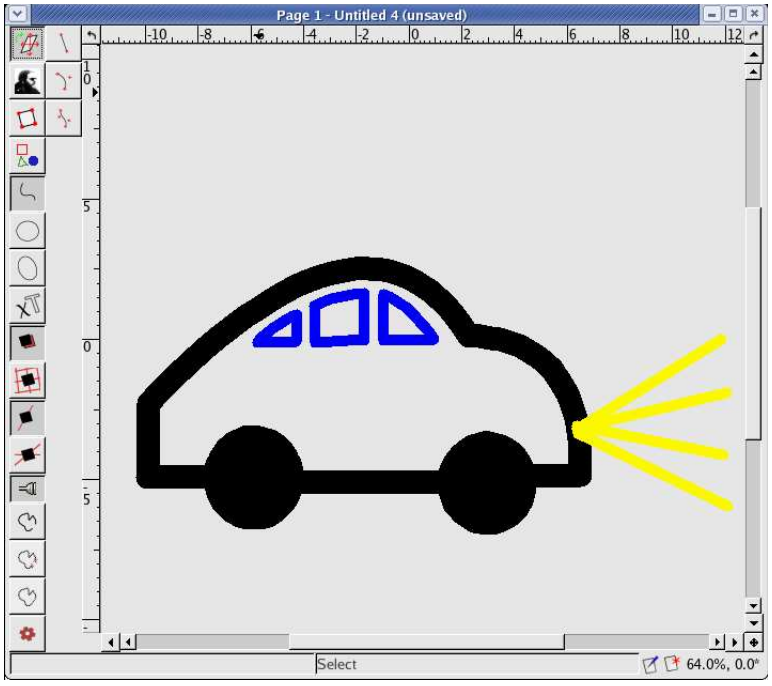
Picture 16: The car chassis.

Continue and draw a segment, a quadratic Bézier curve (the roof) and an elliptic arc defined by 3 points (the front end). For this elliptic arc, set the property “conic” to “cw” or “ccw” to get this picture:



Picture 17: The car hood.

Now select both wheels and set their fill color to black (do not forget to set alpha to 1 to make the fill color opaque). Draw some segments for light rays, change their color to yellow, thicken the line to 3 mm. Finally, draw the windows with blue lines:



Picture 18: The finished picture.

3.6 Snap – introducing geometric dependencies

3.6.1 What is snap? What is it good for?

In the View toolbar, there are yet some unexplored icons. Their purpose is to modify the *snap* settings. When creating new objects, you usually determine some point positions by clicking; and snap may cause the point to be aligned to a significant object nearby. We say that the aligned point is *snapped*. There are several snap modes. If no snap mode is switched on, the chosen point always becomes exactly the position where you clicked. The modes are:

- **Snap to hangers** – a hanger is a significant point (marked by a blue triangle when needed, see Section 4.1 [Anchors and hangers], page 27 for more info.) Hangers are, for example, the end points of a segment, the center point of a circle, etc. When this mode is switched on, the click positions are aligned to a nearby hanger, if there is such.
- **Snap to grid** – if you switch this mode on, a grid appears and the click positions are aligned to grid points. The grid dimensions can be modified in the Settings window (see Section 5.1.7 [Global Settings], page 35).
- **Snap to lines** – aligns the click positions to curves and creates parametric points if needed.
- **Snap to intersections** – aligns the click positions to intersections of curves. The intersection does not have to exist as a regular object; it is computed and created automatically.

The snap modes are independent on each other and can be switched on and off or combined arbitrarily. If you switch at least two modes on, then the closest object of all is chosen regardless of its type (hanger, grid point, intersection or curve).

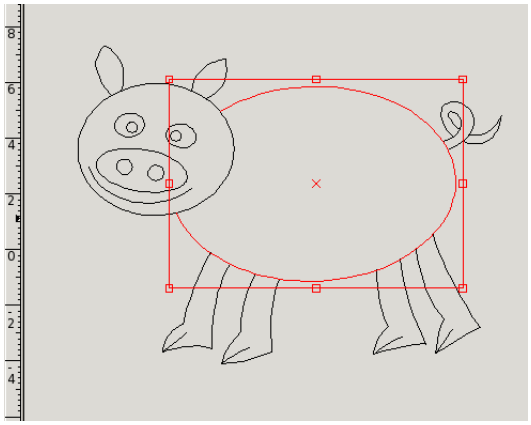
Create dependencies – the last snap icon does not represent any snap mode. It controls the dependency effects of snap. If switched off, the snap just modifies the click position a little without creating any geometric dependency. Otherwise, if there is a significant object near, the point gets stuck to it and anytime the object is moved or transformed, the position of the point is updated as well. The only exception is snap to grid, which generates no dependencies even when "snap to dependencies" is on.

There is a limit of the maximum distance between the original point and the snapped one. By default, the limit is ten pixels and can be changed in the Settings window (see Section 5.1.7 [Global Settings], page 35). Naturally, the physical distance limit depends on the current zoom as well.

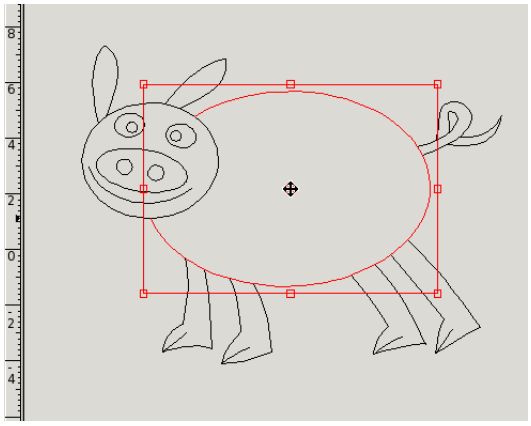
Note: It is not a very good idea to combine snap to hangers with snap to lines. If the target hanger is positioned on a curve (which it usually is), the point might get snapped on the curve very near the hanger but not on the hanger itself.

Note: When trying to transform objects created with some of the snap modes on, you might have encountered an error message like "This selection cannot be transformed." The reason for such an error are the geometric dependencies. For example, if you snap the start point of a segment to the end point of another segment, you cannot move the dependent one. But you can move the other one or both of them. If you need to release the object from dependencies, use the Anchor rehang tool (described Section 3.6.3 [Anchor rehang], page 19).

In the next two pictures, you can see an image containing geometric dependencies before and after transformation. The body of the pig is moved and all snapped objects are recomputed accordingly. Notice that, when moving the ellipse, the dependent objects change their shape, not simply move. That is because they are determined by other points which are not being transformed.



Picture 19: A pig before the transformation.



Picture 20: The pig after the transformation.

3.6.2 Fifi

Fifi is a secondary mouse cursor which indicates the snap position. By default, it is disabled; you can enable it in the Settings window (see Section 5.1.7 [Global Settings], page 35).

Note: The implementation of Fifi is currently somewhat experimental. It is not recommended to use Fifi for large images when snap to intersections is on; VRR has no optimizations for computing so many intersections so far, which makes it considerably slow.

3.6.3 Anchor rehang

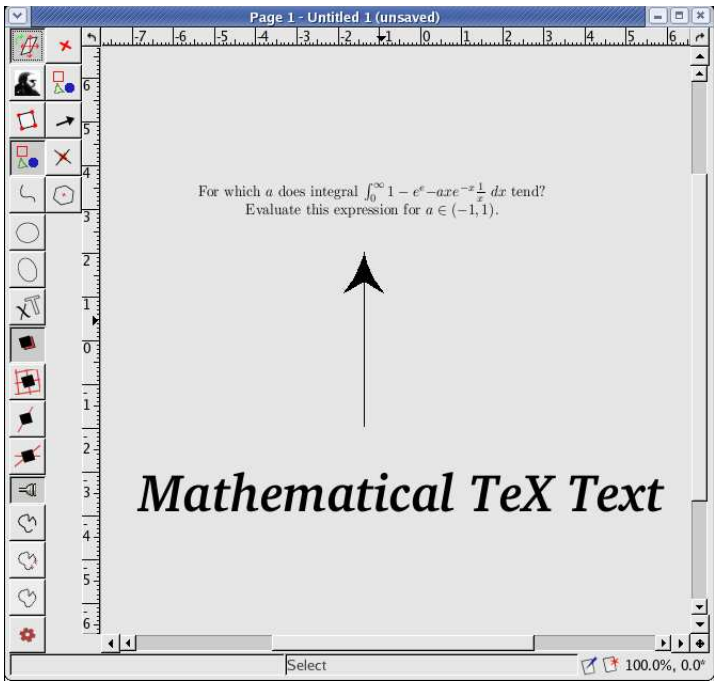
Suppose you have created an object and you want to reposition some of the points you chose when creating it. You can do that using the Anchor rehang mode. First, select an object; its "source" points appear as green triangles (*anchors*, see Section 4.1 [Anchors and hangers], page 27). By clicking any of them and clicking the destination hanger, you reposition the appropriate anchor and modify the object's geometric dependencies accordingly.

When choosing the new position of the anchor, snap works normally and you can use all the snap modes (see Section 3.6.1 [What is snap? What is it good for?], page 18).

3.7 Creating texts

3.7.1 Texts – Overview

Apart of all other features, VRR has a sophisticated system for working with text. In addition to common text objects, it allows you to create text objects typeset by TEX and make them part of your image. You can create almost anything from a short math expression to several paragraphs of a text. Type1 PostScript fonts and TrueType fonts are fully supported (including exports to various formats) in ordinary text objects.



Picture 21: A simple example of TEX and ordinary text objects.

Creating an ordinary text object and creating a TEX-text object differ slightly, but basically they are the same. Both start with choosing a reference point to specify the text object location. An editor window is then opened and you fill in the source text and all the desired options. The "align", "relative-shift" and "absolute-shift" properties control the position of the reference point with regard to the bounding box of the resulting text. The property values are described in Section 5.3.4.5 [Texts], page 44.

For example, if you set the refpoint values to: refpoints-relative, bbox-relative, 0.5, 0.5, 0.0 and 0.0, the text will be positioned in such a way that center point of its bounding box equals the reference point.

The text area of the editor shows the source text of the text object. You can edit it directly, load it from a file or save it to a file (both in the character encoding set in your locale). You can also edit the text with an external editor (like `vim`, `emacs`, etc). To run the editor, make sure you have set its name (in the Settings window, see Section 5.1.7 [Global Settings], page 35) and press the "Edit with external editor ..." button; when you have finished editing, save all changes and finish your editor. VRR looks frozen while running the external program as it is waiting for it to exit.

Note: This does not work properly with editors that fork their process at startup (`gvim`, for example).

Any changes you have made to the source text take effect after pressing the "Refresh" button or by pressing the *Ctrl* + *Enter* keyboard shortcut.

For ordinary text objects, the source text is the same as the text displayed except for newlines which are drawn as spaces; if you want to create text labels containing more than one line, use a TEX-text object. For TEX-text objects, the source text is a TEX source code. You can use any TEX commands you like. The only restriction is that VRR cannot properly handle TEX output containing more than one page.

You can try, for example, this source text:

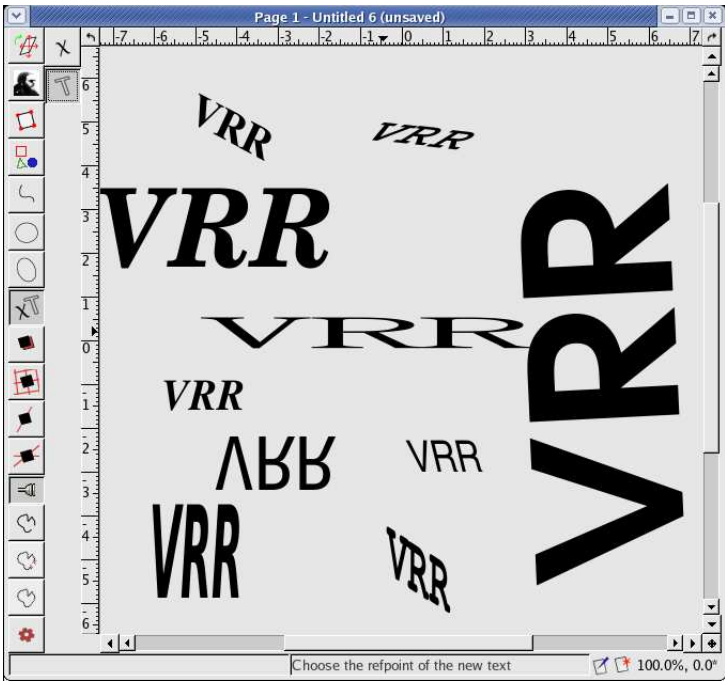
```
My first \TeX{} code in VRR is an integral:
$\int_3^5 x^2 {\rm d}x$
```

or an even nicer one:

```
\def\VrR{V\kern-0.15em\lower0.5ex\hbox{R}\kern-0.15emR}
My first \TeX{} code in \VrR{} is an integral:
$\int_3^5 x^2 {\rm d}x$
```

In case of the ordinary text, there are additional widgets in the editor for choosing the font and font size. In the font combo box, you can see the list of all installed fonts.

Note: The text pixmaps for large text sizes consume a lot of memory. Therefore, if you set the font size to a too large number or scale the text object to be too large, the text object becomes invisible to prevent the size of the pixmap to exceed the limits. But the text object itself is not destroyed – when you make the text smaller again, the text reappears.



Picture 22: Many text objects with different fonts and transformations.

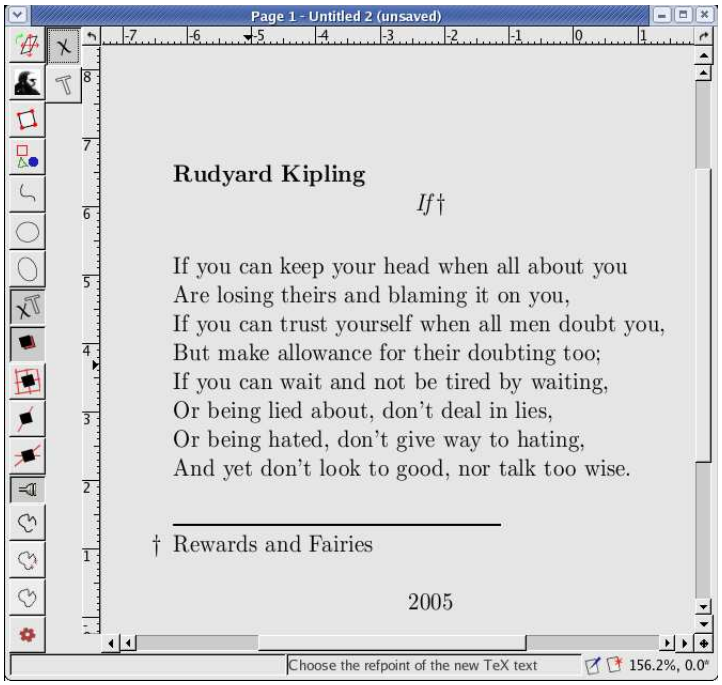
3.7.2 Texts – Examples

We will create some more examples of mathematical TEX text. Switch into the TEX text creating mode and write the following source text:

```
\noindent
\centerline{For which $a$ does the integral
$\int_0^\infty 1 - e^x - axe^{-x}{1 \over x} {\rm d}x$ converge?}
\centerline{Evaluate this expression for $a \in (-1, 1)$.}
```

Create an ordinary text "Mathematical TeX text" and place it below the created TEX text. Proceed with a vertical segment and an arrow on the top of it (the arrows can be found in the View toolbar in the "Points, decorations" category). The resulting image looks like the one in the beginning of this section.

The following example demonstrates some of the large amount of TEX features which you can use in VRR as well:



Picture 23: A more sophisticated TEX text.

The TEX source code for the previous image is:

```
\hsize=8cm\hoffset=1cm
\vsize=6cm\voffset=1cm
\parindent=0pt
\footline={\hfil 2005\hfil}
\leftline{\bf Rudyard Kipling}
\centerline{\it If\rm\footnote{\dag}{Rewards and Fairies}}
\vskip 0.5cm
\obeylines
If you can keep your head when all about you
Are losing theirs and blaming it on you,
If you can trust yourself when all men doubt you,
But make allowance for their doubting too;
If you can wait and not be tired by waiting,
Or being lied about, don't deal in lies,
Or being hated, don't give way to hating,
And yet don't look too good, nor talk too wise.
```

3.7.3 TeX tutorials

The previous examples of TEX source code are far from cover all the useful TEX macros and features. It is surely beyond the scope of this book to teach you how to use TEX. To learn more about TEX, you might want to try some of the following books:

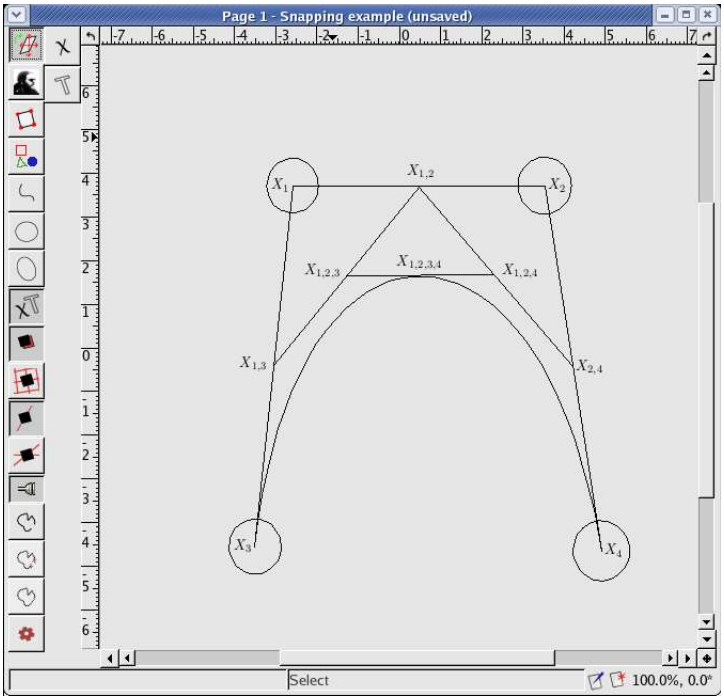
Donald E. Knuth: The TEXbook, Addison-Wesley, reprinted 1993 – if you want to become a TEX guru, this is the right book for you.

Paul W. Abrahams, Karl Berry: Tex for the Impatient, Addison Wesley 1990 – a simplified TEXbook. Each concept and command is explained in a separate entry with many helpful examples. It is available online at http://tug.org/ftp/tex/impatient/.

Michael Doob: A Gentle Introduction to TEX – a very easy-to-understand tutorial for beginners. It is available online at http://ctan.tug.org/tex-archive/info/gentle/gentle.pdf.

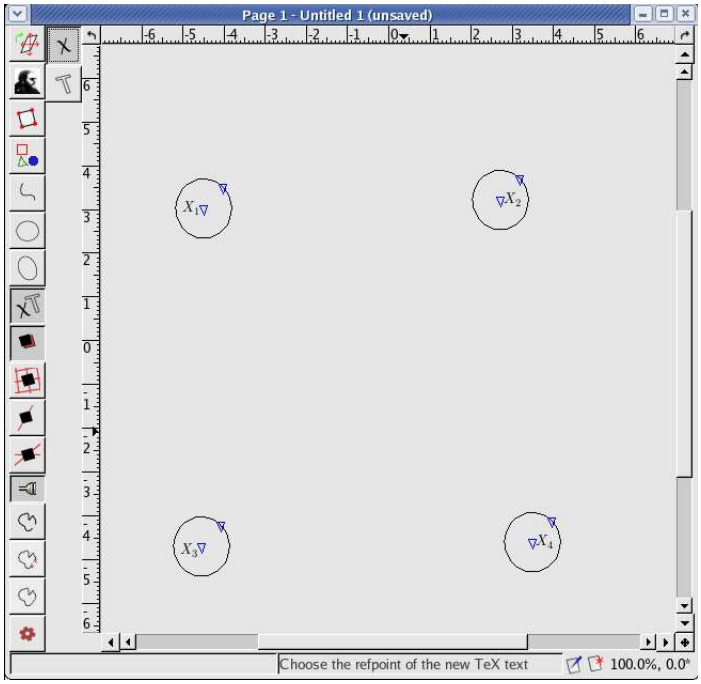
3.7.4 Texts and Snap – the Bézier subdivision example

Now that you can create and transform objects, use snap and create TEX texts, you are ready for a more sophisticated example. We will create this picture:



Picture 24: The Bézier subdivision example.

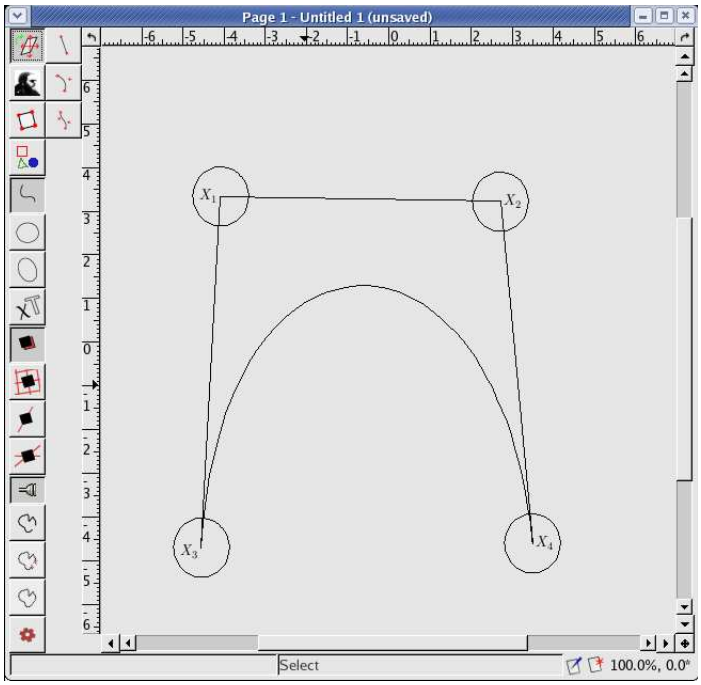
First, create four circles. Then create four TEX texts with the TEX source codes `$X_1$`, `$X_2$`, `$X_3$`, `$X_4$`, and snap them to the center points of the circles. If you keep the texts' default properties unchanged, the texts will overlap the center points. To avoid that, modify the absolute-shift-x property in the Property editor (the meaning of the absolute-shift-x property is quite obvious). Set the values so that in the Anchor rehang mode you do not see the center point hangers being overlapped by the texts.



Picture 25: Four circles with labelled center points.

Now create segments connecting the points X_1 and X_2, X_1 and X_3, X_2 and X_4 so that the end points of the segments are snapped to he circle center points.

Create a quadratic Bézier curve with the control points X_3, X_1, X_2, X_4:

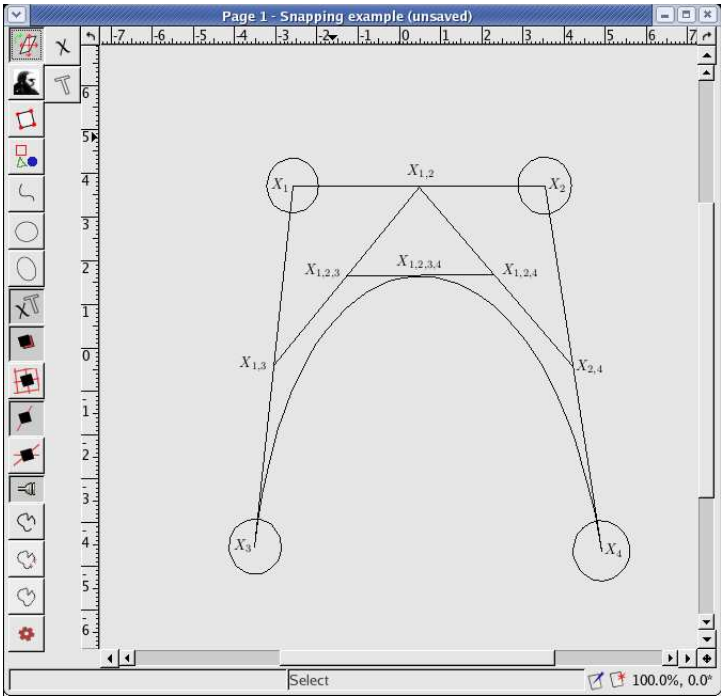


Picture 26: The four circles connected by segments and a Bézier curve.

Create parametric points in the middle of the Bézier curve and the segments (switch on the line snapping mode, choose the "Points, decorations" category in the left toolbar and click the "New point" icon in the right toolbar) and click on the line. The parametric point is created,

now change its property “parameter” in the Property window (see Section 3.5 [Modifying the properties], page 14) to value “0.5”. Repeat that for each segment and also for the Bézier curve. Describe these points with TEX labels: $X_{1,3}$, $X_{1,2}$ and $X_{2,4}$ for the center points of the segments, and $X_{1,2,3,4}$ for the Bézier curve's parametric point (with the source code `$X_{1,2}$`, `$X_{1,2,3,4}$` and similar).

Now create segments connecting the parametric points, create parametric points in their middles and link them by a segment (the middle label of this segment is $X_{1,2,3,4}$). The resulting image is displayed in the following picture:



Picture 27: The resulting Bézier subdivision example.

To see how the created dependencies work, try to move some of the circles and watch the rest of the image.

3.8 Groups and paths

Groups

In VRR, you can also work with *groups*. A group is a set of graphic objects which behaves as one object from outside. It is selected or transformed as a whole, or you can open a View on it and manipulate its contents individually.

The group structure of a page is hierarchical: a group can contain multiple groups and/or individual graphic objects in it. To gather objects in a group, select them and choose "Group/Group selected". To break a group and make the objects independent again, choose "Group/Ungroup". In the Universe browser window you can see the group hierarchy. To open a View for a group, use the "Group/Open in new view" command. When viewing the contents of a group, the contents objects of superior groups become invisible and you can only see the contents of the group and its subgroups.

The z-order

The main purpose of groups is to make your image structured. The objects inside each group have a certain order; new objects are placed on top. This order is called the *z-order* and can be edited using the "Edit/Move up", "Edit/Move down", "Edit/Move to front", "Edit/Move to back" commands or the `U`, `D`, `Shift + U`, `Shift + D` keyboard shortcuts, respectively. In a superior group, the group moved in the z-order as a whole. By grouping some objects, you group them together in the z-order as well.

Paths

A special case of a group is a *path*. In addition to the features of groups, a path has its own style properties which are used for all its contents. Thus, if you have, for example, a path with blue stroke-color, all lines are drawn in blue regardless of the colors of the individual lines. Moreover, a path can be filled – the fill-color property controls the fill color of the entire path.

To create a path, switch the path mode on by clicking the icon  "Path mode on/off" in the View toolbar. You can see that some of the icons become disabled as not all graphic objects can be contained in a path. You can now create segments, Bézier curves and arcs. Start with a first one and then continue; each object sticks to the end point of the previous one automatically, because the path must be continuous. When finished, you can either end the path with the  command or with the  command. The latter one completes the path with an enclosing segment. In both cases, you can continue creating another path or switch the path mode off by clicking the  icon again.

See Section 4.2.1 [Groups], page 29 and Section 4.2.2 [Paths], page 29 for more detailed description.

3.9 Controlling VRR from the command line

All the editing actions accessible from GUI (and even some more) can be performed via a command line. VRR has a text console  which accepts commands in the Scheme programming language. The available data types and functions are described in Chapter 6 [Scheme], page 46.

The more detailed description of the Scheme console can be found in Section 5.1.10 [The Scheme console], page 38.

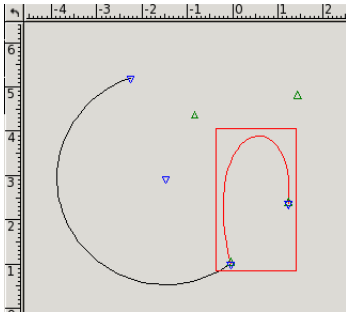
4 The Anatomy of the Universe

This chapter explains the anatomy of pictures: what are hangers, anchors, graphic objects, what are geometric dependencies, how it all works together, ... You will learn many details about the objects you have met in the previous chapter.

When we talk about "the universe", we mean all the objects reachable for you; they are documents, pages, graphic objects etc. and are visible in the Universe Browser window. We now start with the smallest elements you can manipulate with and gradually build the whole universe in the bottom-up direction. The most important elements are the graphic objects. They contain anchors and hangers. The graphic objects are grouped into groups, pages and documents.

4.1 Anchors and hangers

Since you have learned how to create new graphic objects (in Section 3.1 [The first simple graphic objects], page 5), you know that every object is determined by a certain number of point positions (plus more data, like properties etc). The number of point positions is fixed for each object type. On the other hand, the object generates some significant points where other objects can be snapped.



Picture 28: A simple image with hangers and anchors displayed.

In VRR terminology, we talk about *hangers* and *anchors*. Anchors are marked by green triangles and symbolize the "input" points, whereas hangers are marked by blue triangles and symbolize the "output" points. In the picture above, you can see a circular arc and a selected cubic Bézier curve (in the Anchor rehang mode, so all the hangers and the anchors of selection are visible). Notice that the hangers of an object can but do not have to correspond with its anchors. The circular arc has three anchors: the start point, midpoint and end point. The start point and end point have their matching hangers, but midpoint does not. There is a center point hanger instead. The start point anchor of the Bézier curve is hanging on the end point hanger of the arc.

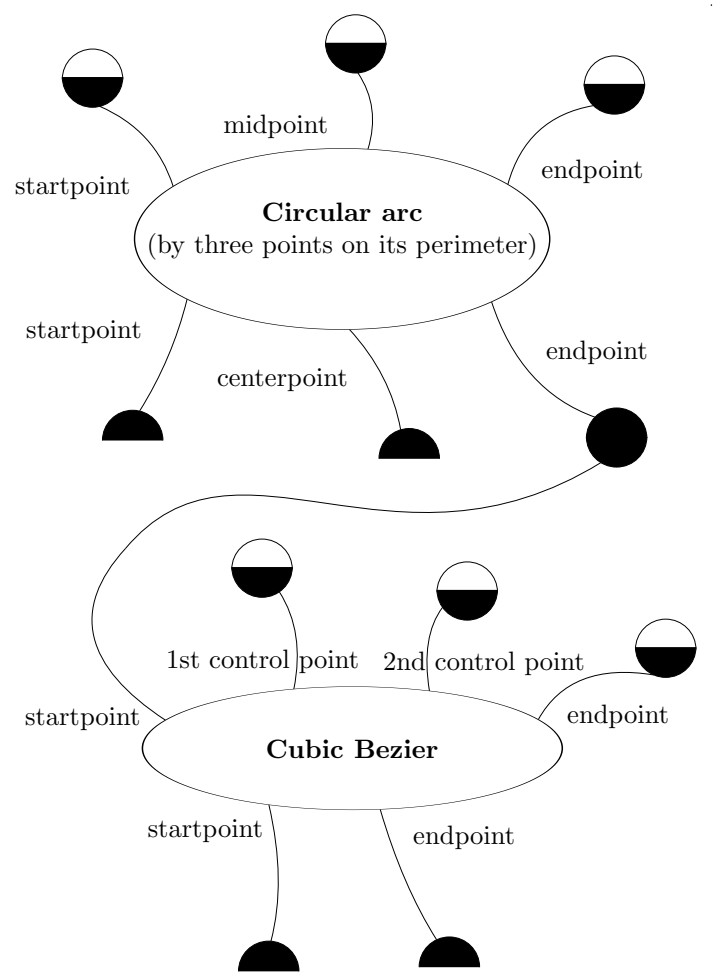
Naturally, the number of on object's anchors does not have to be equal to the number of hangers. As observed in the picture, the Bézier curve has four anchors, but two hangers only. An anchor must hang on exactly one hanger. A hanger can hold arbitrarily many anchors.

Optional: Apart from position hangers, the objects can have curve hangers as well. For example, a parametric point is not associated with any particular point position; it is determined by the whole curve and a parameter (a number between zero and one which defines the point's relative distance from the start point). Therefore, its anchor does not hang on any position hanger, it hangs on a curve hanger of a curve. So it is not precisely true that a Bézier curve has two hangers only, it has a curve hanger as well.

When creating a new graphic object and choosing the point positions, you determine the positions of the object's anchors. If you use the "Snap to hangers" mode (see Section 3.6.1 [What is snap? What is it good for?], page 18) and click near a hanger, you *hang* the anchor on the hanger and make them associated. We say that you created *geometric dependency*. From now on, the dependent object will be recomputed automatically after any change of its anchors' hangers.

Note: Naturally, because of the recomputing order, you cannot create circular dependencies – you cannot have an object A hanging on B, B hanging on C and C hanging on A. Also, you cannot have an object's anchor hanging on a hanger of the object itself. Such dependencies are forbidden and when trying to create them, you will not be successful.

The following picture shows the dependency diagram of the previous image:



Picture 29: The dependency diagram of the previous image.

The big ellipses represent the two graphic objects. Anchors are represented by half-circles pointing up, hangers are half-circles pointing down. You can see the start point anchor of the Bézier curve hanging on the end point hanger of the arc.

Mouse-clicks

Notice that every anchor hangs on a hanger, even those that are not snapped. The white half-circles stand for special hangers called *mouse-clicks*. Mouse-clicks do not belong to any graphic object, they exist freely inside the page. Moreover, they are not connected to any fixed point position, so the object hanging on mouse-clicks only is independent and can be transformed freely.

The mouse-clicks are created and destroyed automatically, you do not have to do that explicitly. But they are full-value hangers and they are marked with blue triangles as well. You can even rehang anchors on them using the Anchor rehang mode (Section 3.6.3 [Anchor rehang], page 19).

4.2 The group tree of a page

4.2.1 Groups

A group is an ordered set of graphic objects which behaves as one object from outside. It is selected or transformed as a whole, or you can open a View on it and manipulate its contents individually. The selection in a group is independent of other groups.

The group structure of a page is tree-like. A group can contain multiple groups and/or individual graphic objects in it. The order of the objects inside a group is known as the *z-order*, the newly created objects are positioned to the top (the start) of the group. A group is atomic in the z-order of its superior group. To learn how to edit the z-order, see Section 3.8 [Groups and paths], page 26.

The group hierarchy is independent of the geometric dependencies. An object can be dependent on another object in a totally different group. Anyway, we do not recommend you to create very complicated dependency relationships which cross the group hierarchy many times, as you might easily lose track about what you have created and which object is dependent on which one.

A page is a special case of a group. The page's default group is called the *top-level group*. When you create a blank page, it contains the top-level group only. Top-level groups are not visible in the Universe browser and you cannot delete, ungroup or manipulate them.

4.2.2 Paths

A special case of a group is a *path*. In addition to the features of groups, a path has its own style properties which are used for all its contents. Thus, if you have, for example, a path with blue stroke-color, all lines are drawn in blue regardless of the colors of the individual lines. Moreover, a path can be filled – the fill-color property controls the fill color of the entire path (in the "even-odd" fill style). Apart from the fill style, the way a path is drawn differs slightly from the way VRR would draw all the individual objects: the lines are connected together and line caps are used for path end points only.

The objects in a path obey strict geometric dependency requirements. The start point hanger of each object must hang on the end point anchor of the previous one. Therefore, when you try to use the Anchor rehang tool (see Section 3.6.3 [Anchor rehang], page 19) on the contents of a path, it usually fails as the dependency rules cannot be (even temporarily) broken). Not all objects can be contained in a path – the suitable objects are segments, Bézier curves and circular/elliptic arcs.

4.3 Documents and pages

As you already know, every page contains a tree-like group hierarchy. It was also already mentioned (in Section 3.3 [Basic actions], page 10) that each page has its own undo history and that the edit actions performed on a page are independent on actions performed on other pages.

You can undo and redo actions in a page without any effect on other pages' undo histories, regardless of the global historical order of edit actions. These undo histories are called "local".

To store edit actions not connected to contents of any page, VRR has the "global" undo history. It can be opened in the Undo History window (Section 5.1.4 [The Undo History Window], page 33), too. It contains creating, deleting, selection changes of documents and pages (not the pages' contents), changes of global settings etc.

Note: When you create a page, edit its contents and then undo a global action, you delete the page. But the contents of the page stay untouched; if you redo the global action, you restore the page together with all its graphic objects. The same holds for deleting pages, and creating and deleting documents works similarly.

5 The Anatomy of the Graphical User Interface

This chapter explains all the important features of the GUI – all windows, the context, all graphic objects that can be created and the procedure how to do it.

5.1 Windows

5.1.1 The Main Window

When you run the application, the main window opens. It allows you to do some basic actions not connected to any particular document (create new documents, load documents from files, open some windows as Universe Browser or Undo History, set global VRR settings or open help files). These windows will be described in the following chapters.

Picture 30: The main window.

By closing the main window you terminate the program. Before the exit, it asks you if to save the unsaved files if there are any.

5.1.2 The View

The purpose of the view window is to display the contents of a document's page or group. Each view displays one page/group, while a page/group can be displayed in several independent views. All changes performed to the contents are displayed in all views at once. All views displaying the same page/group can be used interchangeably. By closing the view, you do not delete the page/group nor the document containing it, you only close the view. After creating a new document or opening the existing one (with at least one page), a view for the first page is opened.

When displaying the contents of a group, the objects inside the superior groups are invisible; you can only see the contents of the group and its subgroups.

The drawing area of the view is potentially infinite. However, the scrollbars always scroll over a limited area. The area is enlarged automatically to exceed the bounding box of all objects a bit. Or you can enlarge it manually by clicking the scrollbar arrows repeatedly when the border of the area is reached – the scrollbar arrows allow you to move even farther than the current editable area borders and thus you have the area enlarged.

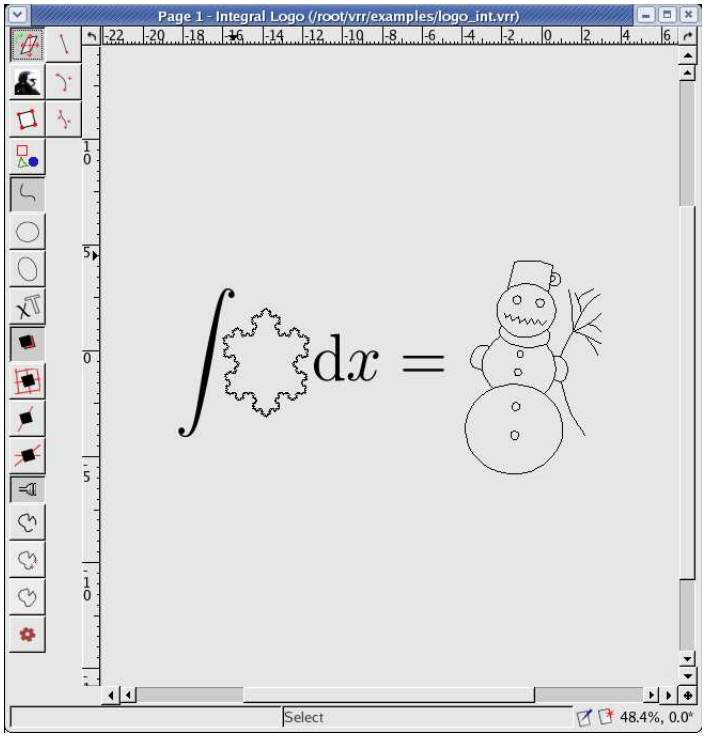
Some settings of the view can have effect on the displayed image without changing the image's properties actually. These are zoom and rotation. Their current values are displayed in the bottom right corner of the view. Zoom can be changed by scrolling the mouse wheel, by pressing the `-`, `=` keys or the `-`, `+` numpad keys. Pressing the `R` key resets both rotation and zoom.

By dragging the middle mouse button you move the image in the view or the view over the image – choose one of the possibilities in the Settings window Section 5.1.7 [Global Settings], page 35, the "Panning: Drag the image" toggle button. If set to true, you drag the image and the mouse button stays pinned to a point position. Otherwise, you drag the view and the image moves in the opposite way than the mouse cursor does.

You can also center a chosen point in the view using `Ctrl` + middle mouse button click. The right mouse button opens the context menu and the left mouse button does all the rest – almost all editing actions are done by clicking the left mouse button.

The buttons in the toolbars (on the left side of the view window) can be used to create new graphic objects, set snap modes etc. In the left toolbar, there are icons that represent an icon category. By clicking them you expand the contents of the category into the right toolbar.

The left part of the status bar occasionally shows some messages, usually error messages. In the right part you can see hints regarding the current editing actions (what to do in the particular editor state, see Section 5.3 [The mechanism of creating new graphic objects], page 39.) The icon in the bottom right corner indicates that the page has been modified since the last save. The indicates that the page is currently being edited (is part of the context; the same icon can be observed in the Universe browser).



Picture 31: The View window.

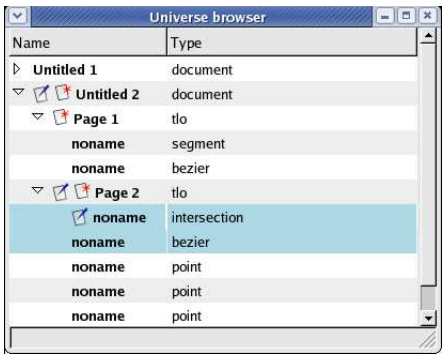
In the corners between scrollbars and rulers, there are four buttons. The upper buttons rotate the view. The bottom right button opens the View Navigator window containing the preview of the whole page. The green rectangle in the preview represents the currently visible region of the page in the view. You can move it and change the visible region of the view accordingly. Enter new values of zoom and rotation to adjust preview. The Reset button sets zoom to 100.0% and rotation to 0 degrees.

5.1.3 Universe Browser

When we talk about "the universe", we mean the set of all existing objects – documents, pages, graphic objects etc. The Universe browser window shows you the tree structure of the universe. For each object, the name and type are displayed. On the left side of the name, there is an expander arrow and icons which appear from time to time. The icon indicates that the object is part of the context (see Section 5.2 [The context], page 39). The icon indicates that the particular document or page has been modified since the last save.

The left mouse button performs selection in the style similar as in Gimp: *`Shift + click`* adds to selection, *`Ctrl + Click`* removes from selection, while a single click clears the current selection

and selects the clicked objects. But the selection is cleared in the current scope only – if you click an object in a group, then all the other objects in that group are unselected but the rest of the universe is unchanged.



Picture 32: The Universe browser window.

The selected objects are marked with a color. The program has two different selections – selection of "namespace objects" (documents and pages) displayed in pink and selection of graphic objects which is blue. The selections are orthogonal, hence changed independently. Also, the selections in distinct groups and pages are independent.

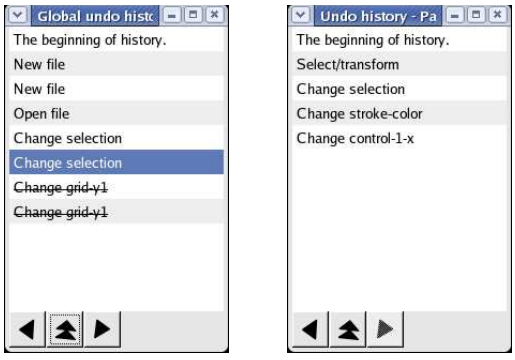
The **right mouse button** opens the pop-up menu containing actions available from View's menu as well. In addition, there are the "Global undo" and "Global redo" menu items which operate the global undo history (undo history actions performed on documents and pages, not on the contents of any page).

5.1.4 The Undo History Window

The Undo history window can show the "local" undo history (of a page) or the "global" undo history containing the actions not connected to the contents of any page. It shows the list of all actions that have been done or undone. The undone actions are striked trough. After a new action, all undone items are deleted.

The toolbar buttons have the following meaning:

- undo the last not undone action in the list
- jump to the selected undo item
- redo the first undone action in the list



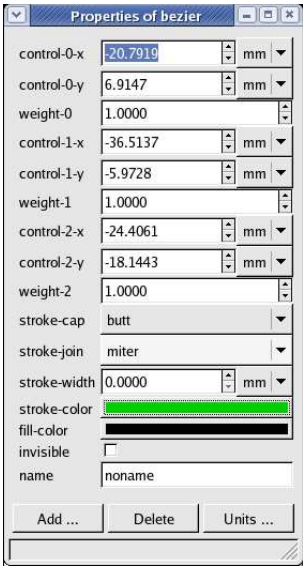
Picture 33: The global and local undo history windows.

The number of items in undo history and the memory they consume are limited by user settings Section 5.1.7 [Global Settings], page 35). Therefore the first items of the list can be removed occasionally.

Note: When creating a new graphic object, you might notice that each click causes a new undo item to appear. That makes the "Step back" feature possible – it actually undoes the last partial undo operation. When the object is created, all partial undo items are merged into one.

5.1.5 The Property Window

Each object, according to its type (segment, circle etc.), contains some specific properties, which can be – in general – changed. The Property window allows you to view and change the values of an object's properties.



Picture 34: The Property window.

The attempt to change some property values may fail for various reasons, mainly the geometric dependencies. If you try to change the coordinates of the center point of a circle determined by three points on its perimeter, it fails. In that case, the value remains unchanged. The value of some properties is also limited according to their *subtype*. The subtype determines the meaning of the property – length, angle, color etc.

Properties with dimensions are associated with a dimension unit. By setting the unit, you modify the display value of the property, not the property itself (the internal value stays the same). You can also create and use your own units (click on "Units ..." button to open the Unit manager Section 5.1.8 [The Unit Manager], page 37).

For some properties, there is a more specialized editor, like the color editor, text editor (see Section 5.1.6 [The Text Editor], page 35), filename editor etc. In that case, there is a button in the property window that opens the editor.

You can also create, edit, and delete your own properties; that can be useful for making some notes or in script processing. To add a property, press the "Add ..." button and fill in the name (the key), type, subtype and value of this property. Note that the property key must be unique. To delete some properties, select them by clicking their names in the Property window (or unselect by clicking once more). Then press the "Delete" button. Note that not all properties can be deleted; some of them are important for geometric and graphic features of the object.

But you can always delete the properties you created yourself. Even these property actions can be undone and redone using the local undo/redo.

The context property window The Property window has two slightly different forms: the *Context* one and the one connected to one object only. The context window can be opened via "Windows/Property window". It shows the properties of the current context object (see Section 5.2 [The context], page 39) or of the current selection, if there is any. When editing the selection properties, only the properties common for all selected objects are displayed. All changes are done to all selected objects. The property values of the first selected object are displayed.

5.1.6 The Text Editor

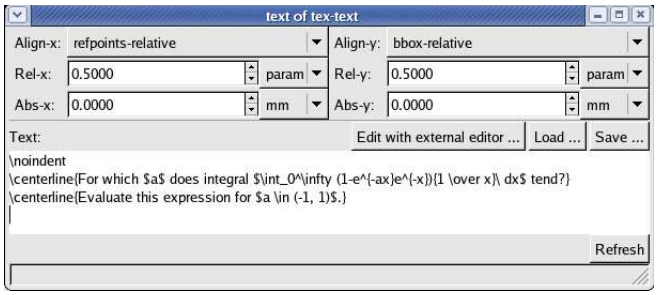
When creating a new text (or TEX text), you first specify a hanger – location of the text reference point. Then the Text editor is opened. The Text and TEX text editors differ slightly, but basically they are the same. The main text area shows the source text of the text object. You can edit it directly, load it from a file or save it to a file (both in the character encoding set in your locale). You can also edit the text with an external editor (like `vim`, `emacs`, etc). To run the editor, make sure you have set its name (in the Settings window, see Section 5.1.7 [Global Settings], page 35) and press the "Edit with external editor . . ." button; when you have finished editing, save all changes and finish your editor. VRR looks frozen while running the external program as it is waiting for it to exit.

Note: This does not work properly with editors that fork their process at startup (`gvim`, for example).

Any changes you have made to the source text take effect after pressing the "Refresh" button or by pressing the ***`Ctrl` + `Enter`*** keyboard shortcut.

The "align", "relative-shift" and "absolute-shift" properties control the position of the reference point with regard to the bounding box of the resulting text. For description of their meaning see Section 5.3.4.5 [Texts], page 44.

In case of the ordinary text, there are additional widgets for choosing the font and font size. In the font combo box, you can see the list of all installed fonts.



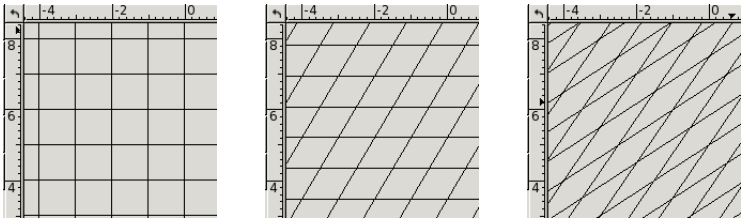
Picture 35: The Text editor window.

5.1.7 Global Settings

The Global settings window enables you to customize the behaviour of VRR. You can find it under the "Windows/Global settings" menu command or the icon. We now describe the settings in detail.

- **Snap tolerance (in pixels)** – determines the snap tolerance used for snap (Section 3.6.1 [What is snap? What is it good for?], page 18). It is the maximum allowed distance between the original point and the snapped one.

- **Grid x1, y1, x2, y2** – the vectors (x_1, y_1) and (x_2, y_2) define the View grid. All grid points are their integer linear combinations. The default is a square grid, but you can set it to any possible grid defined by two linearly independent vectors. If you set two dependent vectors, one of them is ignored and a perpendicular vector of the same size as the other one is used instead.
- **Ruler x-resolution, ruler y-resolution** – the default horizontal and vertical resolution for the rulers. They define the base section length in the rulers.
- **Panning: drag the image** – determines whether to move the image or the view (using the middle mouse button in the View drawing area). If set to true, you drag the image and the mouse button stays pinned to a point position. Otherwise, you drag the view and the image moves in the opposite way than the mouse cursor does.
- **Enable Fifi** – turns on/off the VRR Fifi. When Fifi is turned on, the blue secondary cursor is be displayed and indicates the snap position. See Section 3.6.2 [Fifi], page 19.



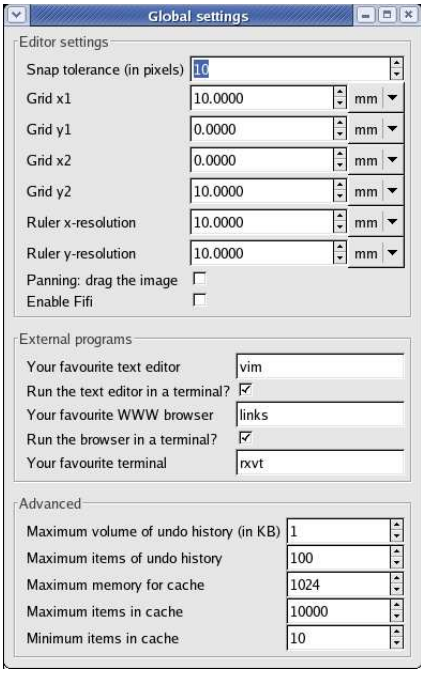
Picture 36: Some of the possible grid settings: the default grid, a "triangular" grid, and a diamond grid.

- **Your favourite text editor** – choose the editor you want to use for text editing (see Section 5.1.6 [The Text Editor], page 35). You can omit the full path and write the program name only.
- **Run the text editor in a terminal?**
- **Your favourite WWW browser** – choose the browser to view Help with. You can omit the full path and write the program name only.
- **Run the browser in a terminal?**
- **Your favourite terminal** – in what terminal should the chosen external text editor and the help browser eventually run?
- **Maximum volume of undo history (in KB)**
- **Maximum items of undo history** – the maximum number of undo items per page.

Important: The editor needs several undo items to enable the "Step back" feature. If you set the maximum number of items to a very small number (less than five), the editor will not work properly.

- **Maximum memory for cache** – the settings of cache for geometrical computations.
- **Maximum items in cache**
- **Minimum items in cache**

There is no "Refresh" button. All changes take effect immediately or after pressing ***Enter*** (for text entries and spin buttons). The settings are saved automatically when exiting VRR and loaded during startup.



Picture 37: The Settings window.

5.1.8 The Unit Manager

The Unit manager window shows the list of defined units and also allows you to edit them. The window consists of three parts:

On the top, there is a edit box with list of unit quantities like length, angle or reference. Choose the one you want to work with (view, create, delete, or edit). The list below will be refreshed – only units of chosen quantity will be displayed.

The biggest part of the window is the list of displayed units. The unit's name is the name displayed in unit lists in property editors. The multiplier (against a certain unit) defines the unit's value. For example, if you want to create a unit for kilometers, set the multiplier to 1000 (against meters). The multiplier is then recomputed against an internal unit value.

The bottom part contains 4 buttons:

- **New** – adds a new unit. The unit name must be unique.
- **Edit** – allows you to change the values of an existing unit except for quantity. If you want to change the quantity, delete the unit and recreate it again.
- **Delete** – deletes the selected unit. You cannot delete the default unit.
- **Set as default** – set the selected unit to be the default for the chosen quantity. The default unit is displayed in property editors for properties for which you have not changed the unit manually. It cannot be deleted.

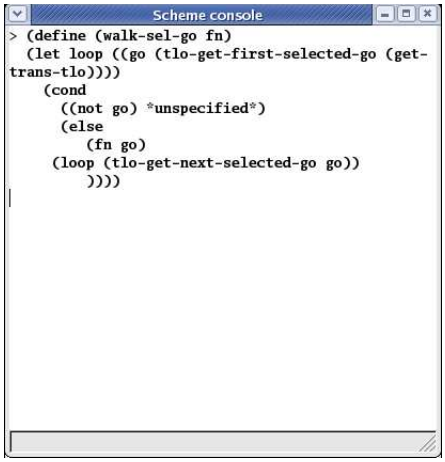
5.1.9 The Plugin Manager

The Plugin manager allows you to load or unload plugins. All you need to load a plugin is to click the "Load . . ." button and choose its location in the file browser. In the table, all loaded plugins and their functions are displayed.

Some plugins can be unloaded (for such, the "Unload" button is enabled). To unload the selected plugin, press the "Unload" button.

5.1.10 The Scheme console

All the editing actions accessible from GUI (and even some more) can be performed via a command line. VRR has a text console which accepts commands in the Scheme programming language. The available data types and functions are described in Chapter 6 [Scheme], page 46.



Picture 38: The Scheme console.

The console text area is composed of two parts – the active text buffer and the history. The active buffer is a place in which you can edit a new command. In history, old commands and VRR answers are disaplayed. The current buffer is displayed in bold letters while the history is displayed in regular letters. The console supports all standard GTK text editing features (but only the active buffer is editable). In addition there are these keystrokes:

Enter if the current cursor position is in the active buffer, the console tries to execute the entered command. If there are missing closing parentheses in the entered command (the text in the active buffer), missing-parenthesis-prompt is displayed ("... "). Otherwise the command is executed and together with VRR output becomes a part of history. The new command prompt is displayed ("> ") and active buffer is empty. In both cases the cursor is moved at the end of the active buffer.

Enter if the current cursor position is in the history, the console copies the old command (which is near the cursor) at the end of active buffer. So, if you want to reuse a previous command, move the cursor up to it and paste it by pressing *Enter*.

Shift + *Enter* in the active buffer just enters newline and does not try to execute anything.

Ctrl + *S* starts incremental search forward and *Ctrl* + *R* starts incremental search backward. In incremental search, as you are typing, the newly entered letters are added to the sought text and the cursor jumps to places where the sought text is found. By pressing *Ctrl* + *S* or *Ctrl* + *R* repeatedly, you move the cursor to next occurences of the sought text.

BackSpace during incremental search moves to an old position or removes the last entered letter from the sought text.

Each command corresponds to one undo history item (see Section 5.1.4 [The Undo History Window], page 33) and is executed in a transaction of the active page.

5.1.11 The Clipboard

The Clipboard window is, in fact, a View (see Section 5.1.2 [The View], page 31). You can work with it in the same way as in a View (create graphic objects, manipulate them, . . .). The only difference is that you are modifying the clipboard. But beware, editing the clipboard contents together with copy, cut and paste operations can be a bit confusing. For example, if you want to paste the clipboard contents in a page, only the selected objects will be pasted. If you copy some objects into the clipboard, the current clipboard contents will be deleted.

5.2 The context

During your work with VRR, some objects or windows become implicitly significant from time to time: the object you clicked last, the top-level window or the current selection. So, when performing an action, you do not have to specify all the subjects explicitly. The collection of these significant things is called the *context*. It consists of:

- the current selection
- the last clicked object
- the current group
- the current page
- the current window, if there is such (usually a view, which determines its page and the page's father document)

The selection has always the highest priority. If the selection is empty, the other objects are used. Almost all actions, like keyboard shortcut commands, menu commands etc. operate with the current context. For example, if you press `Ctrl+Z`, which means Undo, the last action is undone in the context page.

The context objects are indicated with the  icon. The icon can be found in the View and in the Universe browser.

You might have noticed that the menus can be torn off. The torn-off menu is not connected to any document or page, it works with the current context as well. You can watch the menu items being disabled and enabled again as the context changes.

5.3 The mechanism of creating new graphic objects

Creating a new graphic object works similarly to a finite automaton: You set the starting state (e.g. "Create a segment") by clicking an icon the the View toolbar, and then choose the desired arguments of the operation (in this case, two hangers), step by step. By pressing the `BackSpace` key, you return one step back, by pressing `Esc`, you cancel the creation process and delete the GO that is being created. Once you choose the desired argument (usually by clicking an object with the left mouse button), in case of success the editor moves to another state and allows you to choose another argument. In case of a failure (e.g. when creating a circumscribed circle of three points and having chosen the first two points as equal) the editor reports an error and asks you to choose the last argument again. The error can occur for various reasons: when trying to create a circle circumscribed to three collinear points, or move a fully dependent object, for example. When the creation process is finished, the editor returns to the starting state again so that you can create another GO of the same type.

The GO creating state is global for the whole VRR. You can operate on one page at one time (but you can work in any of the page's Views interchangeably). Setting some other page as the context page resets the process and returns to the starting state. Any action incompatible with the current GO creating action, the process is cancelled and the editor returns to the starting state, as well.

Almost all other actions are incompatible with GO creating operations, like selection, menu commands, ... The most important exception are the snap buttons – you can change snap settings even during the creation of a new graphic object. Thus you can snap the first point of an object to a hanger, the second one to the grid and the third one to a line, for example. See Section 3.6.1 [What is snap? What is it good for?], page 18.

The interface for creating a new GO is contained in the view. Toolbar buttons switch between editor states, by clicking the drawing area you position the points as arguments of the current operation, select objects, move the transformation gadgets and thus transform objects, etc. The argument required in the current state is described in the right part of the status bar.

The editor uses the undo history to enable the "Step back" feature and needs to make sure that no other actions interfere with its own. That is the reason why almost all menu commands interrupt the current operation. Moreover, the editor needs to have several undo items enabled (in the Global settings, see Section 5.1.7 [Global Settings], page 35). If you set the maximum number of undo history items to a too small number (say, three), the "Step back" feature will work in a very odd way.

The editor has the following kinds of states/modes: the Select/Transform mode, the Santiago's transform mode, the Anchor rehang mode and the go creating modes. We now describe these modes in detail:

5.3.1 The Select/Transform mode

In the Select/Transform mode, you can select objects and transform the selection. By *Shift* + click you add an object to selection and by *Ctrl* + click you remove it from selection. Clicking an object without any of *Shift* or *Ctrl* pressed clears the selection and makes it the only selected object in the page. The keyboard shortcuts work similarly for rectangular selection. Instead of plain clicking, press the left mouse button, by dragging it define a rectangular region, and release. This modifies the selection with all objects inside the region. In addition, the "Edit/Select all" command (*Ctrl* + *A*) selects all objects in the page and the "Edit/Clear selection" command (*Shift* + *Ctrl* + *A*) clears the selection. You can also clear the selection by clicking the drawing area far enough from any object.

You can see a red rectangle bounding all selected objects. Also, the selected objects are drawn in red instead of their real color. This helps you to recognize which objects are selected and which are not. On the red selection bounding box, there are small squares of various colors. These are the transformation gadgets, each gadget stands for a certain transformation. While holding the *Shift* key, you can see different transformation gadgets. The meaning of the gadgets is described in Section 3.4.1 [Transforming using the Select/Transform tool], page 11.

In case that the bounding box of selected objects is (almost) non two-dimensional, only some gadgets are shown and only some transformations are applicable. These are **move**, **scale** and **rotate** for bounding boxes which are almost horizontal and vertical lines, and **move** for point bounding boxes.

5.3.2 The Santiago's transform mode

The Santiago's transform tool enables you to do all affine transformations in a special way. Having the *Shift* key pressed, click to position the three transformation crosses. The first click places the first cross, the second one places the second cross etc. By clicking an already positioned first or third cross, you remove it. By clicking the second cross, you toggle it between the states blue, red, removed. If a cross with lower number is removed, the crosses with higher number are disabled (gray).

Now drag a cross to transform. For all crosses, the transformation is computed in such a way that the point from the original cross position becomes the point at the new cross position.

The first (magenta) cross is the gadget for move.

The second (blue/red) cross is the gadget for resize/rotate, which are all possible linear transformations (may be combined together). The first cross is the fixed point.

The third (green) cross does the skew. The former crosses (and the line connecting them) are the fixed points. This enables you to do all affine transformations.

Note: The transformation in the Santiago's tool is computed according to the relative positions of the three crosses among one another. It does in no way depend on their absolute position in the image (or with regard to the selection bounding box), which might seem somewhat surprising at first.

5.3.3 Anchor Rehang mode

This mode allows you to rehang anchors (defined in Section 4.1 [Anchors and hangers], page 27). First, choose the owner object of the anchor by selecting it. You can see its anchors appear as green triangles, and all hangers as blue triangles. By clicking any of them and clicking the destination hanger, you reposition the appropriate anchor and modify the object's geometric dependencies accordingly.

5.3.4 GO Creating Modes

Almost all editor icons represent states for creating new GOs. In the left toolbar, there are some icons which represent icon groups (Points and decorations, Bézier curves, etc). Click the group icon to expand all the icons contained in the group into the right toolbar.

In the descriptions of the GO creating modes, we also give a list of type, subtype, hanger and anchor terms (canonical names). You can ignore them unless you are interested in the Scheme; in that case, they give you useful information about the GOs.

5.3.4.1 Points and decorations

-  **Point**
-  **Decoration point**

The Decoration point represents a point with a certain shape. The shape is defined by the number of vertices – zero means a circle, two a segment, three up to one hundred a polygon and more than one hundred a circle again. The size of the decoration point is determined by the value of the "radius" property.

Note: The decorator point is resistant to transformations. This is very useful for decorating many point which should look the same, for example, vertices of a graph.

-  **Arrow**

The arrow is a transformation-resistant decoration, too. It should be positioned on a curve and it adjusts its direction according to the curve's tangent in the snap position. Its appearance can be adjusted by changing the property values (in the Property editor).

Each arrow has four important points - the front point (where the hanger is located), two side points and the back point. By setting up the "arrow-alignment" property, you can specify how the arrow's rotation should be computed. We can align the arrow to the derivation in the front point or force the back point to be in the intersection of the curve and the side points' center line. The second possiblity is useful especially for a very rounded curve. The final rotation can be adjusted with "rotation".

The front shape of the arrow is controlled by the "arrow-angle" property (half of the angle between the front point and side points), "arrow-length" (distance between that points),"arrow-front" (shape type) and "front-curvature". The property "back-distance" determines the angle between the back point and side points and the "arrow-back" property determines the back shape of the arrow.

• **Intersection point**

To create an intersection point, choose two curves whose intersection you want to create. If do not intersect or cease intersecting after a transformation, the intersection point will position itself on a somehow chosen position (the last well-defined intersection position, for example).

Type: **intersection**
Subtype: **intersection**
Anchors: **curve-1**, **curve-2**
Hanger: **center**

• *n*-**gon**

The *n*-gon generator creates a regular *n*-gon with the given number of apices (a closed path of segments). When you switch to this mode, an entry appears in the editor status bar and lets you enter the desired number of apices. Then you choose the center point and the position of one apex.

5.3.4.2 Bézier curves

Bézier curves used in VRR are rational Bézier curves. For each control point of the curve a rational weight is defined (1, by default) which influences the shape of the curve as well. The weights are the object's properties and can be modified in the Property editor.

• **Segment**

Creates a segment defined by the start point and the end point.

Type term: **segment**
Subtype term: **segment**
Hanger term: **curve**, **start**, **end**
Anchor term: **start**, **end**

• **Quadratic Bézier curve**

Creates a quadratic Bézier curve defined by three control points.

Type: **bezier**
Subtype: **quadratic-bezier**
Hangers: **curve**, **controlpoint-1**, **controlpoint-3**
Anchors: **controlpoint-1**, **controlpoint-2**, **controlpoint-3**

• **Cubic Bézier curve**

Creates a cubic Bézier curve defined by three control points.

Type: **bezier** Subtype: **cubic-bezier** Hangers: **curve**, **controlpoint-1**, **controlpoint-4** Anchors: **controlpoint-1**, **controlpoint-2**, **controlpoint-3**, **controlpoint-4**

5.3.4.3 Circular arcs

In fact, circles and circular arcs are ellipses and elliptic arcs. When being created, they are set to be the circular special cases; but, they do not differ from elliptic objects in anything more significant then just a few property values. Therefore, circles have two radii and other properties which might seem useless.

See Section 5.3.4.4 [Elliptic arcs], page 43 for full description.

• **Circular arc defined by three points on its perimeter**

Creates a circular arc defined by three points on its perimeter. The three points must not be collinear.

• **Circular arc defined by the center point and radius**

Creates a circular arc defined by the center point and radius. The radius, the end point parameters, etc. can be modified in the Property editor.

• **Circle defined by three points on its perimeter**

Creates a circle defined by three points on its perimeter. The three points must not be collinear.

• **Circle by the center point and a point**

Creates a circle defined by the center point and a point on its perimeter.

• **Circle by the center point and radius**

Creates a circle defined by the center point and radius. The radius can be modified in the Property editor.

5.3.4.4 Elliptic arcs

The ellipse is just a special case of an elliptic arc; it can be changed into an elliptic arc just by changing the "conic" property. The "conic" property is quite important and very useful. It controls the type of the arc. The available values depend on the object type. The values for all types are:

- **start-entire** – a closed arc
- **start-dif** – an arc defined by the "start" parameter with "dif" defining the arc length

The available values for arcs with at least one point on the perimeter:

- **point-entire** – a closed arc
- **point-dif** – an arc starting from the start point with "dif" defining the arc length

The available values for arcs with three points on the perimeter:

- **ccw** – an arc connecting the three given points counterclockwise
- **cw** – an arc connecting the three given points clockwise
- **smaller** – the smaller one of the two arcs connecting the start point and the end point and having the third point on the perimeter of the whole circle/ellipse
- **bigger** – the larger one (dtto)
- **middle** – the arc connecting the start point and the end point via the center point
- **opposite** – the opposite to "middle"

• **The smallest ellipse defined by three points**

Creates the smallest (in area) ellipse determined by three points on its perimeter. The points must not be collinear.

Type: **elliptic-arc**
Subtype: **ellipse-by-3-points-smallest**
Hangers: **curve**, **start**, **end**, **center**
Anchors: **point-1**, **point-2**, **point-3**

• **Ellipse defined by three points, rotation and eccentricity**

Creates an ellipse defined by three points on its perimeter, the rotation and eccentricity. The points must not be collinear. The rotation and eccentricity properties can be edited in the Property editor.

Type: **elliptic-arc**
Subtype: **ellipse-by-3-points-rotation-eccentricity**
Hangers: **curve**, **start**, **end**, **center**
Anchors: **point-1**, **point-2**, **point-3**

• **Ellipse defined by two foci and a point**

Creates an ellipse defined by the foci and a point on its perimeter. The two foci points must not be equal.

Type: **elliptic-arc**
Subtype: **ellipse-by-2-foci-point**
Hangers: **curve**, **start**, **end**, **center**
Anchors: **focus-1**, **focus-2**, **point**

• **Ellipse defined by the center point, a point and eccentricity**

Creates an ellipse defined by the center point, a point on its perimeter and eccentricity. The eccentricity property can be edited in the Property editor.

Type: **elliptic-arc**
Subtype: **ellipse-by-center-point-rotation-eccentricity**
Hangers: **curve**, **start**, **end**, **center**
Anchors: **center**, **point**

• **Ellipse defined by the center point and radii**

Creates an ellipse defined by the center point and the two radii. The radii can be edited in the Property editor.

Type: **elliptic-arc**
Subtype: **ellipse-by-center-2-radii-rotation**
Hangers: **curve**, **start**, **end**, **center**
Anchors: **center**

• **Elliptic arc defined by the center point and radii**

Creates an elliptic arc defined by the center point and the two radii. The radii can be edited in the Property editor.

5.3.4.5 Texts

The position of the text objects is defined by a reference point (hanger) and several properties. The "align" property determines the reference point position with regard to the text bounding box. The "relative-shift" and "absolute-shift" properties determine additional adjustments of the position.

Each text label has several special points useful for aligning. The left reference point is usually a point on text's baseline near the left edge of the leftmost letter and is taken from the used font. We also define the right reference point which is exactly on the right edge of text's bounding box in the current VRR's version.

First, the label is aligned according to values of "align" and "relative-shift" properties. The result is then translated by "absolute-shift" and transformed with a stored linear transformation (rotation, scale or skew) around the hanger point.

The possible values of "align-x" (the horizontal align) are:

- **refpoints-relative** – The hanger is placed horizontally between the two reference points with the x coordinate of "relative-shift" as the parameter.

- **refpoints-left** – The parameter is replaced with 0.
- **refpoints-center** – The parameter is replaced with 0.5.
- **refpoints-right** – The parameter is replaced with 1.
- **bbox-relative** – The hanger is placed horizontally between the edges of the text's bounding box.
- **bbox-left** – The parameter is replaced with 0.
- **bbox-center** – The parameter is replaced with 0.5.
- **bbox-right** – The parameter is replaced with 1.

The possible values of "align-y" (the vertical align) are:

- **baseline** – The hanger is placed vertically on the text's baseline.
- **bbox-relative** – The hanger is placed vertically between edges of the text's bounding box.
- **bbox-bottom** – The parameter is replaced with 0.
- **bbox-center** – The parameter is replaced with 0.5.
- **bbox-top** – The parameter is replaced with 1.

-  **TEX text**

Creates the TEX text object determined by the chosen reference point. A text editor is then opened on the created object (see Section 5.1.6 [The Text Editor], page 35 for further details and the window description).

Type: **tex-text**
Subtype: **tex-text**
Anchor: **point**
Hangers:

-  **Text**

Creates the text object determined by the chosen reference point. A text editor is then opened on the created object (see Section 5.1.6 [The Text Editor], page 35 for further details and the window description).

Please note that the text objects do not support newlines which are drawn as spaces; if you need to create text labels containing several lines of a text, use a TEX-text object instead.

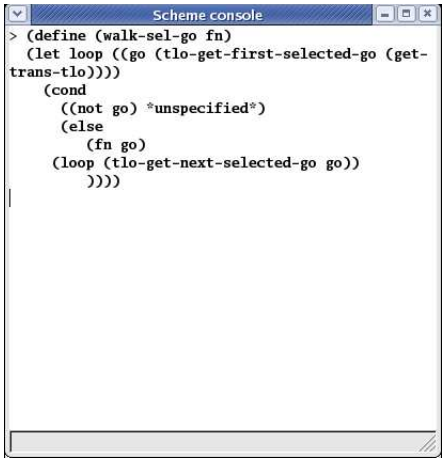
Type: **text**
Subtype: **text**
Anchor: **point**
Hangers:

5.3.5 Snap settings

The  icons control the editor snap settings. They are described in Section 3.6.1 [What is snap? What is it good for?], page 18.

6 Scheme

The `guile-vrr` interface and Scheme console enable you to control the program by Scheme commands. Except for GUI and image viewing, you can access all the program functionality, such as image creating, loading, editing, saving, exporting or importing.



Picture 39: The Scheme console.

This book shows you only some basic examples of Scheme code but does not teach you Scheme systematically. To learn about the Scheme programming language, you might want to try some of the following:

C. Candolin: Scheme Tutorial – a basic tutorial for beginners. It is available online at `http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/schemetutorial.html`.

H. Abelson, G. J. Sussman, J. Sussman: Structure and Interpretation of Computer Programs, MIT Press; available at `http://mitpress.mit.edu/sicp/full-text/book/book.html`).

Revised Report on the Algorithmic Language Scheme – a defining description of the programming language Scheme (`http://www.schemers.org/Documents/Standards/R5RS/HTML/`).

6.1 VRR Scheme data types

VRR proxy is a Scheme object used for representing VRR entities. Proxies are compared using the `eq?` command: if `p1` and `p2` are proxies of the same entity, then `(eq? p1 p2)` returns true. There are four kinds of proxies: `obj`, `go`, `anchor`, and `hanger`, and they are associated with the appropriate VRR entities of the same name. `obj` proxies are classified by their type. There are two types of them: `document` and `page`. `go` proxies can be classified by their type and subtype.

6.2 VRR Scheme functions

In the following sections, there will be many function descriptions. On the first line of each description, there is a 'function prototype' – a function name and formal arguments enclosed in parentheses. If some part of function name is `TYPE`, `SUBTYPE`, `ANCHOR` or `HANGER`, then there is a set of functions which can be obtained by replacing the string with the appropriate terms (for example, `make-SUBTYPE` is a shorthand for function names `make-segment`, `make-quadratic-bezier` and so on).

On the second line, there is a 'type signature'. For each argument, there is a permitted type (the name of the type or type signature enclosed in parentheses in case of a functional argument).

The '+' character is used to union two type sets. Valid values are Scheme type names, VRR type names (proxy, o, obj, go, anchor, hanger and terms of types and subtypes), 'any' (means anything), 'false' (#f) and 'specific' (for specific permitted values, which depend on the function). String 'unspecified' is valid only as a return value (in case that the return value has no meaning). Characters '[' and ']' are used to specify optional arguments (and their types), '...' can be used to specify that the structure of arguments is described in function description.

6.3 Functions for VRR types

`(proxy? obj [kind [type [st]]])`
any [symbol [symbol [symbol]]] → boolean
If no additional arguments are given, is returns whether `obj` is a VRR proxy. Otherwise it also checks the given type information – whether obj kind is `kind`, obj type is `type` and obj subtype is `st`.

`(proxy-kind proxy)`
proxy → symbol
Returns the kind of `proxy`.

`(proxy-type proxy)`
proxy → symbol + false
Returns the type of `proxy`.

`(proxy-subtype proxy)`
proxy → symbol + false
Returns the subtype of `proxy`.

6.4 Creation of objects

There are several definitions of functions for object creation (called also constructors). All return a proxy for the created object. They are usually named by a pattern `make-SUBTYPE`, they need different arguments, so they are described separately. `obj` constructors must be called in a `meta-transaction` and are `n-parent`-less, `go` constructors must be called in a `transaction` and their `g-parent` is `g-root` of the `page` of the current `transaction`.

`(make-document)`
→ document
A constructor for `obj document`.

`(make-page)`
→ page
A constructor for `obj page`.

`(make-common-document name)`
string → document
A modified constructor for `obj document`. The newly created document is named by `name` and connected under the `universe`.

`(make-common-page name doc)`
string document → page
A modified constructor for `obj page`. The newly created page is named by `name` and connected under the `doc`.

`(coords x y)`
real real → hanger

A constructor for a free hanger (also known as a mouse-click). The created hanger coordinates are (`x`, `y`).

`(coords-c p)`
complex → *hanger*
A constructor for a free hanger (also known as a mouse-click). The created hanger coordinates are (`(real-part p)`, `(imag-part p)`).

`(make-point h)`
hanger → *go*
A point constructor. The created GO's `point` anchor is hung on the `h` hanger.

`(make-segment h1 h2)`
hanger hanger → *go*
A segment constructor. The created GO's `start` anchor is hung on the `h1` hanger, the `end` anchor is hung on the `h2` hanger.

`(make-ellipse-by-center-2-radii-rotation c r1 r2 rot)`
hanger real real real → *go*
A constructor for the appropriate `go`. The created GO's `center` anchor is hung on the `c` hanger. `r1` is used as the major radius of the ellipse. `r2` is used as the minor radius of the ellipse. Radii must be positive. `rot` (must be between 0 and 2π) is used as the rotation the of ellipse, where zero rotation means that the major axis is horizontal.

`(make-ellipse-by-2-foci-point f1 f2 p)`
hanger hanger hanger → *go*
A constructor for the appropriate `go`. The created GO's `focus-1` anchor is hung on hanger `f1`, anchor `focus-2` is hung on hanger `f2`, anchor `point` (representing any point on the ellipse) is hung on hanger `p`.

`(make-ellipse-by-3-points-smallest p1 p2 p3)`
hanger hanger hanger → *go*
A constructor for the appropriate `go`. The created GO's anchors `point-1`, `point-2` and `point-3` are hung on hangers `p1`, `p2` and `p3`, respectively. The created ellipse is the smallest (in area) ellipse containing the three points represented by the anchors.

`(make-ellipse-by-3-points-rotation-eccentricity p1 p2 p3 rot ecc)`
hanger hanger hanger real real → *go*
A constructor for the appropriate `go`. The created GO's anchors `point-1`, `point-2` and `point-3` are hung on hangers `p1`, `p2` and `p3`, respectively. `rot` (must be between 0 and 2π) is used as the rotation of the ellipse, where zero rotation means that the major axis is horizontal. `ecc` (must be between 0 and 1) is used as the eccentricity of the ellipse.

`(make-ellipse-by-center-point-rotation-eccentricity c p rot ecc)`
hanger hanger real real → *go*
A constructor for the appropriate `go`. The created GO's anchor `center` is hung on the `c` hanger, the `point` anchor is hung on the `p` hanger (representing any point on the ellipse). `rot` (must be between 0 and 2π) is used as the rotation of the ellipse, where zero rotation means that the major axis is horizontal. `ecc` (must be between 0 and 1) is used as the eccentricity of the ellipse.

`(make-elarc-by-center-2-radii-rotation-2-angles c r1 r2 rot a1 a2 style)`
hanger real real real real real symbol → *go*
An alternative constructor for `go ellipse-by-center-2-radii-rotation`, modified to creation of arcs. The arguments share the semantics with the function `make-ellipse-by-center-2-radii-rotation` and the additional arguments `a1`, `a2` and `style` are the arc information (see Section 5.3.4.4 [Elliptic arcs], page 43).

`(make-elarc-by-3-points-smallest p1 p2 p3 style)`
hanger hanger hanger symbol → go
An alternative constructor for `go ellipse-by-3-points-smallest`, modified to creation of arcs. The arguments share the semantics with function `make-ellipse-by-3-points-smallest` and the additional argument `style` is arc information (See Section 5.3.4.4 [Elliptic arcs], page 43).

`(make-quadratic-bezier p1 p2 p3)`
hanger hanger hanger → go
A constructor for the appropriate `go`. The created GO's anchors `controlpoint-1`, `controlpoint-2` and `controlpoint-3` are hung on hangers `p1`, `p2` and `p3`, respectively. The created bezier is non-rational, i.e. all weights are set to 1.

`(make-quadratic-rational-bezier p1 w1 p2 w2 p3 w3)`
hanger real hanger real hanger real → go
An alternative `go` constructor for `quadratic-bezier`, the created bezier is rational. The arguments share the semantics with function `make-quadratic-bezier`, the additional arguments are weights of the control points.

`(make-cubic-bezier p1 p2 p3 p4)`
hanger hanger hanger hanger → go
A constructor for the appropriate `go`. The created GO's anchors `controlpoint-1`, `controlpoint-2`, `controlpoint-3` and `controlpoint-4` are hung on hangers `p1`, `p2`, `p3` and `p4`, respectively. The created bezier is non-rational, i.e. all weights are set to 1.

`(make-cubic-rational-bezier p1 w1 p2 w2 p3 w3 p4 w4)`
hanger real hanger real hanger real hanger real → go
An alternative `go` constructor for `cubic-bezier`, the created bezier is rational. The arguments share the semantics with the function `make-cubic-bezier`, the additional arguments are weights of the control points.

`(make-tex-text h str)`
hanger string → go
A constructor for the appropriate `go`. The created GO's anchor `point` is hung on hanger `h`, `str` is used as the TEX source text.

`(make-text h str font-family font-style font-size)`
hanger string string string real → go
A constructor for the appropriate `go`. The created GO's anchor `point` is hung on hanger `h`. `str` is used as the text source. `font-family` is the desired font name. `font-style` is the variant (for example, `"regular"` or `"bold"`). `font-size` is the size of the font in millimeters.

`(make-parametric-point go p)`
go real → go
A constructor for the appropriate `go`. The created GO's anchor `curve` is hung on curve hanger of `go`, `p` is used as the position parameter on that curve.

`(make-parametric-point-on-hanger h p)`
hanger real → go
Alternative constructor for `go parametric-point`. The created GO's anchor `curve` is hung on hanger `h` (which must be curve hanger), `p` is used as position on that curve.

`(make-intersection g1 g2 p)`
go go real → go
A constructor for the appropriate `go`. The created GO's anchors `curve-1` and `curve-2` are hung

on curve hangers of `g2` and `g2`, respectively. If there are more intersections of that curves, the one closest to the `p` position on the first curve is used.

`(make-intersection-on-hanger h1 h2 p)`
hanger hanger real → go
An alternative constructor for `go parametric-point`. The created GO's anchors `curve-1` and `curve-2` are hung on hangers `h1` and `h2` (which must be curve hangers), respectively. If there are more intersections of that curves, the one closest to the `p` position on the first curve is used.

`(make-decorator-point h n radius rot)`
hanger natural real real →
A constructor for the appropriate `go`. The created GO's anchor `decorator-point` is hung on hanger `h`. `n`, `radius` and `rot` are used as the number of vertices, radius and rotation, respectively.

`(make-decorator-arrow g p arrow-length rot)`
go real real real →
A constructor for the appropriate `go`. The created GO's anchor `curve` is hung on curve hanger of `go`. `p` is used as the position parameter on that curve. `arrow-length` and `rot` are used as the arrow length and rotation, respectively. The rotation is interpreted as the deviation from the tangent of the curve.

`(make-decorator-arrow-on-hanger h p arrow-length rot)`
hanger real real real →
An alternative constructor for `go decorator-arrow`. The created GO's anchor `curve` is hung on hanger `h` (which must be a curve hanger). `p` is used as the position parameter on that curve. `arrow-length` and `rot` are used as the arrow length and rotation, respectively. The rotation is interpreted as the deviation from the tangent of the curve.

`(make-group)`
→ *go*
A constructor for the appropriate `go`.

`(make-path)`
→ *go*
A constructor for the appropriate `go`.

For some functions, there are aliases (shortened versions):

Function alias	**Function full name**
make-ellipse/c2rr	make-ellipse-by-center-2-radii-rotation
make-ellipse/2fp	make-ellipse-by-2-foci-point
make-ellipse/3ps	make-ellipse-by-3-points-smallest
make-ellipse/3pre	make-ellipse-by-3-points-rotation-eccentricity
make-ellipse/cpres	make-ellipse-by-center-point-rotation-eccentricity
make-elarc/c2rr2a	make-elarc-by-center-2-radii-rotation-2-angles
make-elarc/3ps	make-elarc-by-3-points-smallest
make-bezier/2	make-quadratic-bezier
make-bezier/2r	make-quadratic-rational-bezier
make-bezier/3	make-cubic-bezier
make-bezier/3r	make-cubic-rational-bezier

6.5 The namespace hierarchy and functions

The namespace hierarchy (n-hierarchy) is a structure of pages and documents in the entire VRR.

The N-hierarchy is an ordering over objects of the **obj** kind. This ordering can be also viewed as a set of rooted trees (where nodes are objects of **obj** kind and the root of the tree is the maximum of all nodes in that tree).

One of these trees is more important than others. It is the tree with the object `universe` as the root. This tree represents the accessible objects, other trees are just temporarily detached (or awaiting for attaching). So the n-hierarchy can be often viewed as one tree.

Functions for manipulating with this hierarchy are usually named with `n-` prefix.

Comparisions

These functions do the comparison of n-objects in the n-hierarchy. The root is the maximum.

`(n<=? o1 x2)`
obj obj → *boolean*

`(n>=? o1 x2)`
obj obj → *boolean*

`(n<? o1 x2)`
obj obj → *boolean*

`(n>? o1 x2)`
obj obj → *boolean*

Direct movement

These functions return the appropriate n-neighbours of given a object. Being a sibling is considered to be a reflexive relation.

`(n-parent o1)`
obj → *obj* + *false*

`(n-first-child o1)`
obj → *obj* + *false*

`(n-last-child o1)`
obj → *obj* + *false*

`(n-first-sib o1)`
obj → *obj*

`(n-last-sib o1)`
obj → *obj*

`(n-next-sib o1)`
obj → *obj* + *false*

`(n-prev-sib o1)`
obj → *obj* + *false*

The rest

`(n-leaf? x)`
any → *boolean*
Returns whether the object `x` is an n-leaf. An n-leaf is a member of the n-hierarchy which cannot have children (not just an n-object without n-children), i.e. an object of the **page** type.

`(n-children o1)`
obj → *list*
Returns the list of all n-children of `o1`. In an unspecified order.

`(n-ancestors o1)`
obj → *list*
Returns the list of all n-ancestors of `o1` (i.e. objects that are n-greater than `o1`). In descending n-order.

`(n-descendants o1)`
obj → *list*
Returns the list of all n-descendants of `o1` (i.e. objects that are n-lesser than `o1`). It is ordered like the pre-order deep-first tree walk, so comparable objects are in descending n-order.

`(n-leaves o1)`
obj → *list*
Returns the list of all n-leaves that are n-lesser than `o1`. In an unspecified order.

`(n-set-parent! o1 o2)`
obj obj + *false* → *unspecified*
Sets `o2` as the new n-parent of `o1`. Use `false` for no parent. This must be called in a meta-transaction.

6.6 The group hierarchy and functions

The group hierarchy (g-hierarchy) is a structure of GOs in one page. If a GO is group, then its members are its g-children.

The G-hierarchy is an ordering over objects of the **go** kind in one page. This ordering can be also viewed as a rooted tree (where nodes are objects of the **go** kind and the root the of tree is the maximum of all nodes in that tree), so it is a join-semi-lattice.

Comparisions

These functions do the comparison of g-objects in the g-hierarchy. The root is the maximum.

`(g<=? x1 x2)`
go go → *boolean*

`(g>=? x1 x2)`
go go → *boolean*

`(g<? x1 x2)`
go go → *boolean*

`(g>? x1 x2)`
go go → *boolean*

Direct movement

These functions return the appropriate g-neighbours of a given object. Being a sibling is considered to be a reflexive relation.

`(g-root x1)`
go → *go*

`(g-parent x1)`
go → *go* + *false*

`(g-first-child x1)`
go → *go* + *false*

`(g-last-child x1)`
go → *go* + *false*

`(g-first-sib go)`
go → *go*

`(g-last-sib go)`
go → *go*

`(g-next-sib go)`
go → *go* + *false*

`(g-prev-sib go)`
go → *go* + *false*

The rest

`(g-leaf? x)`
any → *boolean*
Returns whether the object `x` is a g-leaf. A g-leaf is member of the g-hierarchy which cannot have children (not just a g-object without g-children), i.e. a GO which is not of the **group** type.

`(g-children x1)`
go → *list*
Returns the list of all g-children of `x1`. In unspecified order.

`(g-ancestors x1)`
go → *list*
Returns the list of all g-ancestors of `x1` (i.e. objects that are g-greater than `x1`). In descending g-order.

`(g-descendants x1)`
go → *list*
Returns the list of all g-descendants of `x1` (i.e. objects that are g-lesser than `x1`). It is ordered like the pre-order deep-first tree walk, so comparable objects are in descending g-order.

`(g-leaves x1)`
go → *list*
Returns the list of all g-leaves that are g-lesser than `x1`. In unspecified order.

`(g-set-parent! go grp)`
go group → *unspecified*
Sets `grp` as the new g-parent of `go`. It must be called in an appropriate transaction.

Because g-order and z-order is not completely independent, setting the g-parent also affects the z-order. This function tries to minimize the change. The next four functions allow a better control of z-change during the setting of a g-parent:

`(g-set-parent/before go grp g2)`
go group go → *unspecified*
Like `g-set-parent!`, but `go` will be just before `g2` in the z-order (`g2` must be also a child of `grp`).

`(g-set-parent/after go grp g2)`
go group go → *unspecified*
Like `g-set-parent!`, but `go` will be just after `g2` in the z-order (`g2` must be also a child of `grp`).

`(g-set-parent/top go grp)`
go group go → *unspecified*
Like `g-set-parent!`, but `go` will be the minimal possible in the z-order.

`(g-set-parent/bottom go grp)`
go group go → *unspecified*
Like `g-set-parent!`, but `go` will be the maximal possible in the z-order.

6.7 The dependency hierarchy and functions

The dependency hierarchy (d-hierarchy) is another structure of GOs in one page. If a `GO1`'s anchor hangs on `GO2`'s hanger, then `GO1` is d-smaller than `GO2` and `GO1` is a d-child of `GO2`.

The d-hierarchy is an ordering over objects of the **go** kind in one page. This ordering can be also viewed as a directed acyclic graph (where nodes are objects of the **go** type).

`(d-parent go id)`
go symbol → *go*

Returns the `go`'s parent which is accessible through the anchor named `id`.

`(d-children go id)`
go symbol → *list*

Returns the subset of `go`'s children which are accessible through the hanger named `id`.

`(d-all-parents go)`
go → *list*
Returns all parents of `go`.

`(d-all-children go)`
go → *list*
Returns all children of `go`.

`(d-set-parent! g1 k1 g2 k2)`
go symbol go symbol → *unspecified*
Rehangs `g1`'s anchor named `k1` on `g2`'s hanger named `k2`, so `g2` becomes a parent of `g1`. Must be called in an appropriate transaction.

6.8 Anchor-hanger binding functions

Because functions from the d-hierarchy are sometimes awkward to use, there is another set of functions which can be used to obtain similar results. Most of the consequent functions are clear if you know that:

- The connection between GOs, hangers and anchors looks like this:
  go1 – hanger – anchor – go2,
- a go is joined to a set of hangers (each identified with a term),
- a go is joined to a set of anchors (each identified with a term),
- a hanger is joined to exactly one go and set of anchors (without identification, ordered arbitrarily)
- an anchor is joined to exactly one go and exactly one hanger,
- anchor siblings are anchors hanging on the same hanger.

`(go-hanger g id)`
go symbol → *hanger*

`(go-hangers g)`
go → *list*

`(go-anchor g id)`
go symbol → *anchor*

`(go-anchors g)`
go → *list*

`(anchor-first-sib a)`
anchor → *anchor*

`(anchor-last-sib a)`
anchor → *anchor*

`(anchor-next-sib a)`
anchor → *anchor*

`(anchor-prev-sib a)`
anchor → *anchor*

`(anchor-hanger a)`
anchor → *hanger*

`(anchor-go a)`
anchor → *go*

`(hanger-first-anchor h)`
hanger → *anchor*

`(hanger-last-anchor h)`
hanger → *anchor*

`(hanger-anchors h)`
hanger → *list*

`(hanger-go h)`
hanger → *go*

`(h-HANGER g)`
go → *hanger*

Returns the appropriate hanger of GO `g`. It is a shorthand for calling `go-hanger`. For example, `(h-start go)` is equivalent to `(go-hanger go 'start)`.

`(a-ANCHOR g)`
go → *anchor*

Returns the appropriate anchor of GO `g`. It is a shorthand for calling `go-anchor`. For example, `(a-start go)` is equivalent to `(go-anchor go 'start)`.

`(i-HANGER g)`
go → *hanger*

Returns the hanger on which the appropriate anchor of GO `g` is hanging. It is a shorthand for calling `go-anchor` and `anchor-hanger`. For example, `(i-start go)` is equivalent to `(anchor-hanger (go-anchor go 'start))`. `i-` means input.

`(anchor-set-hanger! a h)`
anchor hanger → *unspecified*
Changes the anchor `a` to hang on hanger `h`. This function must be called in an appropriate transaction.

`(set-i-ANCHOR! g h)`
go hanger → *unspecified*

Changes the appropriate anchor of GO `g` to hang on hanger `h`. It is a shorthand for calling `go-anchor` and `anchor-set-hanger!`. For example, `(set-i-start! go h)` is equivalent to `(anchor-set-hanger! (go-anchor go 'start) h)`. `i-` means input. The function must be called in an appropriate transaction.

`(anchor-term a)`
anchor → *symbol*
Returns the term for anchor `a` (identifying that anchor in the corresponding go).

`(hanger-term h)`
hanger → *symbol*
Returns the term for hanger `h` (identifying that hanger in the corresponding go).

6.9 Inter-hierarchy movement

`(go-superior-page g)`
go → *page*
Returns the page to which GO `g` belongs.

`(page-g-root p)`
page → *group*
Returns the root of the g-hierarchy in page `p`.

6.10 Z-order functions

The z-order is an ordering of GOs in one page which represents the depth in which the GOs are situated. So, if GO1 is in front of GO2, then GO1 is z-lesser than GO2.

Comparisions

`(z<=? x1 x2)`
go go → boolean

`(z>=? x1 x2)`
go go → boolean

`(z<? x1 x2)`
go go → boolean

`(z>? x1 x2)`
go go → boolean

Reordering

`(z-move-top! go)`
go → boolean
Moves the `go` on the top in its current group. Returns whether any real movement happened.

`(z-move-up! go)`
go → boolean
Moves the `go` one step up in its current group. Returns whether any real movement happened.

`(z-move-down! go)`
go → boolean
Moves the `go` one step down in its current group. Returns whether any real movement happened.

`(z-move-bottom! go)`
go → boolean
Moves the `go` to the bottom in its current group. Returns whether any real movement happened.

6.11 Selection functions

Information

`(selected? os)`
go → unspecified
Returns weather the given GO is selected.

`(selected-objs)`
obj → list
Returns the list of all selected objects of the **obj** kind.

`(selected-g-children g)`
go → list
Returns the list of all selected g-children of `g`.

`(selected-g-descendants g)`
go → list
Returns the list of all selected g-descendants of `g`. The list is ordered like a pre-order deep-first tree walk, so comparable objects are in descending g-order.

Modification

`(select o1 ...)`
go ... → unspecified
Selects the GOs given as arguments. The function must be called in an appropriate transaction.

`(unselect o1 ...)`
go ... → unspecified
Unselects the GOs given as arguments. The function must be called in an appropriate transaction.

`(select-g-children g)`
go → unspecified
Unselects g-children of given GO. The function must be called in an appropriate transaction.

`(unselect-g-children g)`
go → unspecified
Selects g-children of given GO. The function must be called in an appropriate transaction.

`(unselect-all)`
→ unspecified
Unselects all GOs in the page specified by the current transaction. The function must be called in an appropriate transaction.

6.12 Transformational functions

Functions in this section are used to apply various transformations on selected objects. These functions accept a group argument and transformations are only applied to selected g-children of the chosen group (because to their descendants the application is implicit - transforming a group means transforming all its children).

The second part of this section (functions with the `-active` suffix) is analogic to first part, but the group argument is missing, because it is taken from the active group (the group of the active view), as returned by `(active-group)`. Because this concept exists only with the GUI, these functions are not accessible in the command-line variant of VRR.

General transformations

`(move-selected grp x y)`
group real real → unspecified

Moves the appropriate GOs by `x` in the X axis and `y` in the Y axis.

`(rotate-selected grp cx cy angle)`
group real real real → unspecified

Rotates the appropriate GOs by the `angle` angle. The center of the rotation is `<cx, cy>`.

`(scale-selected grp cx cy sx sy)`
group real real real real → unspecified

Scales the appropriate GOs by `sx` in the X axis and `sy` in the Y axis. The center of the scale is `<cx, cy>`.

`(skew-selected grp f1x f1y f2x f2y fx fy tx ty)`
group real real real real real real real real → unspecified

Skews the appropriate GOs in a such way that the line of the skew is (`<f1x, f1y>`, `<f2x, f2y>`) (everything on that line is not affected by the skew) and the point `<fx. fy>` is transformed to `<tx, ty>`.

`(transform-selected grp a b c d e f)`
group real real real real real real → unspecified

Transforms the appropriate GOs by the transformation matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}.$$

`(remove-selected grp)`
group → unspecified

Removes the appropriate GOs.

`(copy-selected src-grp dst-grp)`
group group → unspecified

Copies the appropriate GOs to `dst-grp`.

Transformations in active view

`(move-active x y)`
real real → unspecified

`(rotate-active angle)`
real real → unspecified
The center of the rotation is in the center of the view.

`(scale-active sx sy)`
real real → unspecified
The center of the scale is in the center of the view, but the axes of the scale are not altered by the rotation of the view.

`(skew-active f1x f1y f2x f2y fx fy tx ty)`
real real real real real real real real → unspecified

`(transform-active a b c d e f)`
real real real real real real → unspecified

`(remove-active)`
→ unspecified

`(copy-active dst-grp)`
group → unspecified

`(duplicate-active)`
→ unspecified
Copies the appropriate GOs to same the group.

6.13 Windows and views

It is even possible to use Scheme to manipulate with VRR windows, namely with Views. For this purpose there is a special data type representing VRR windows. All these functions are not available in the command-line variant of VRR, of course.

General information

`(window? x)`
any → *boolean*
Returns whether the given object is a window.

`(active-document)`
→ *document* + *false*
Returns the active document, i.e. the document of the active page.

`(active-page)`
→ *page* + *false*
Returns the active page, i.e. the page of the active view.

`(active-group)`
→ *group* + *false*
Returns the active group, i.e. the group of the active view.

`(active-window)`
→ *window* + *false*
Returns the active window.

`(active-view)`
→ *view* + *false*
Returns the active view - if the active window is not a view, then it returns false.

View information

`(view? x)`
any → *boolean*
Returns whether the given object is a view.

`(view-group vw)`
view → *group*
Returns the group displayed in the given view.

`(view-page vw)`
view → *page*
Returns the page of the group displayed in the given view.

`(view-width vw)`
view → *natural*
Returns the width of the given view (in pixels).

`(view-height vw)`
view → *natural*
Returns the height of the given view (in pixels).

`(view-center-x w)`
window → *real*
Returns the X coordinate of the image in the center of the view.

`(view-center-y w)`
window → *real*
Returns the Y coordinate of the image in the center of the view.

View manipulation

These functions can be used for changing the transformation between the image and the view (like scrolling and zooming).

`(make-view g)`
group → *view*
Creates a new view displaying the group `g`.

`(view-move vw x y)`
view real real → *unspecified*
Moves (scrolls) the given view by `x` and `y` millimeters in the X and Y axes, respectively.

`(view-rotate vw angle)`
view real → *unspecified*
Rotates the given view around its center by the `angle` angle.

`(view-scale vw scale-x scale-y)`
view real real real → *unspecified*
Scales the given view around its center by the `angle` angle.

`(view-set-center vw x y)`
view real real → *unspecified*
Scrolls-in the given view so that the point with the coordinates `<x`, `y>` becomes the center of the view.

`(view-set-orientation vw center-x center-y scale-x scale-y angle)`
view real real real real real → *unspecified*
Transforms the given view so that the point with the coordinates `<center-x`, `center-y>` becomes the center of the view and the X axis zoom, Y axis zoom and rotation are set to `scale-x`, `scale-y` and `angle`, respectively.

6.14 Propertial functions

Property values can also be accessed from Scheme. Each property value type has some external representation, which is used for setting and getting the value of the property. For real numbers, booleans and strings it is easy, other property value types use a list with symbols (as identifiers of the property value type) in the first position:

`(natural n)`
: where `n` is a natural number.

`(rgb r g b)`
: where `r`, `g`, `b` are values from 0 to 255, the color is solid.

`(rgba r g b a)`
: where `r`, `g`, `b`, `a` are values from 0 to 255.

`(font font-family font-style)`
: where `font-family` and `font-style` are strings.

`(cap-style symbol)`
: where `symbol` is one of: `butt`, `round`, `projecting`

`(join-style symbol)`
: where `symbol` is one of: `miter`, `round`, `bevel`

`(alignment-x symbol)`
: where `symbol` is one of: `ref-left`, `ref-center`, `ref-right`, **ref-relative**, **bbox-left**, `bbox-center`, `bbox-right`, `bbox-relative`

`(alignment-y symbol)`
: where `symbol` is one of: `baseline`, `bbox-left`, `bbox-center`, `bbox-right`, `bbox-relative`

`(arrow-front symbol)`
: where `symbol` is one of: `straight`, `parabolic`

`(arrow-back symbol)`
: where `symbol` is one of: `none`, `straight`, `poly`, `parabolic`

`(arrow-alignment symbol)`
: where `symbol` is one of: `front`, `back`

There exist even more types, but these are the most useful.

For example, to set the fill color to magenta for an object `g`, you should run:

```
(set-property! g 'fill-color '(rgb 127 255 0))
```

`(get-property o name [missing])`
o symbol [any] → *specific*
Returns the value of the property `name` of `o` or (`missing` or false if `missing` is not set) if no such property exists.

`(set-property! o name value)`
obj + *go symbol specific* → *unspecified*
Sets the property `name` of the object `o` to `value`. The property is created if no such property exists.

`(get-conic g)`
elliptic-arc → *list*
Returns the list with arc information (see Section 5.3.4.4 [Elliptic arcs], page 43) about `g`. In the list there is the value of the **conic** property and other associated properties (`start`, `dif`) if they are meaningful.

`(set-conic! g conic ...)`
elliptic-arc symbol ... → *unspecified*
Sets the arc information (see Section 5.3.4.4 [Elliptic arcs], page 43) of `g`. It sets the **conic** property to `conic` and the other associated properties (`start`, `dif`), which should be given as arguments when they are needed.

6.15 Transactional functions

At first, you can skip this section if you only use Scheme commands in console to do some work wit GOs in the active page.

Transactions are a mechanism which allows the grouping of changes to VRR data structures. In case of some problem in one change, all performed changes to the data structures in the current transaction are undone. Changes in the Scheme environment are not affected. Transactions are also used in undo/redo – one successfully passed transaction is one undo history item.

Transactions are somewhat related to exceptions. They both have the same purpose. Transactions are a mechanism of VRR and exceptions are a mechanism of GUILE Scheme. These two mechanisms are cooperating. If there is an unhandled exception inside a transaction, then the transaction fails and an exception is propagated outside. If there is a transaction and it fails from some other reason than an exception, then an exception **transaction-failed** is thrown.

There are several kinds of transactions: Meta-transactions, which are responsible for operations on objects of the **obj** kind. Common transactions, which are responsible for operations on GOs. And finally subtransactions, which can be nested in meta-transactions or common transactions.

Common transactions take an additional argument – a page. So in one common transaction, there can be operations on GOs in one page; for operations on GOs in another page another transaction must be used.

Why was there a notice about skipping at the start of this section? Because the VRR console has a feature of automatic adding common transactions on the active page around the entered commands. So in common usage there is no need to use transactions explicitly. But the explicit usage of transactions is needed in `guile-vrr`.

`(trans-main page thunk [handler [dsc]])`
... → any
Executes `thunk` (a function without arguments) in a transaction at `page` (if `page` is a regular page) or a meta-transaction (if `page` is the universe). `handler` (common GUILE exception handling function) is executed in case of a transaction fail. The default handler raises an exception (or forwards in case that a fail was caused by an unhandled exception). `dsc` is used for transaction description.

`(trans thunk [handler])`
... → unspecified
Executes `thunk` (a function without arguments) in a subtransaction of the current transaction. `handler` (common GUILE exception handling function) is executed in case of a subtransaction fail. The default handler raises an exception (or forwards in case that the fail was caused by an unhandled exception).

`(trans-page)`
→ page + *false*
Returns the page of the current transaction or false in case that no common transaction is active.

`(trans-fail msg)`
string → unspecified
Causes the fail of the current transaction. `msg` is a message with short description of the reason of the fail.

`(undo g)`
page + *universe → unspecified*
Makes global undo (in case that `g` is the universe) or local undo in the page `g` (in case that `g` is a page).

`(redo x)`
go + *universe → unspecified*
Makes global redo (in case that `g` is the universe) or local redo in the page `g` (in case that `g` is a page).

6.16 Miscellaneous functions

`universe`
A constant for the root of the n-hierarchy.

`(lookup name)`
string → go + *obj* + *false*
Returns an obj or GO which has the property **name** set to `name` (or false if no such object exists).

`(load-doc filename)`
string → document
Loads a document from file `filename`.

`(save-doc doc filename)`
document string → unspecified
Saves the document `doc` to file `filename`.

`(save-as-ps doc filename pw ph fit-to-page without-fonts)`
document string real real boolean boolean → unspecified
Exports the document `doc` as PostScript to the file `filename`. The page dimensions are `pw`, `ph` or computed if `fit-to-page` is set. `without-fonts` says whether fonts should be stored in the PostSript file, too.

`(save-as-eps page filename without-fonts)`
page string boolean → unspecified
Exports the page `page` as an Encapsulated Postscript to file `filename`. `without-fonts` says whether fonts should be stored in the EPS file, too.

`(save-as-pdf doc filename without-fonts)`
document string boolean → unspecified
Exports the document `doc` as PDF to file `filename`. `without-fonts` says whether fonts should be stored in the PDF file, too.

`(save-as-svg page filename)`
page string → unspecified
Exports the page `page` as SVG to file `filename`.

`(plugin-call id arg1 ...)`
number any ... → any
Calls a plugin function identified by the ID `id`.

`(load-plugin filename)`
string → unspecified
Loads the plugin `filename`.

7 FAQ

• What is TEX? Do I need to know it to use VRR?

TEX is a typesetting system written by Donald E. Knuth that was "intended for the creation of beautiful books – and especially for books that contain a lot of mathematics". Although TEX text objects are one of the main VRR features, you do not need to know TEX to work with VRR and create valuable and sophisticated images in it.

To learn about the TEX typesetting program, see Section 3.7.3 [TeX tutorials], page 23.

• What is Scheme? Do I need to know it to use VRR?

Scheme is a dialect of the Lisp programming language. in VRR, you need it to write commands in the Scheme console. However, all the important commands are accessible from the GUI, too.

To learn about the Scheme programming language, see the beginning of Chapter 6 [Scheme], page 46 to find a list of links to tutorials.

• What is the purpose of the `.scm` example files? How do I open them?

The `.scm` files contain Scheme scripts (commands usable in the Scheme console). You can either copy and paste them into the console, or load them using the command

```
(load "/write/the/filename/here")
```

in the console. It may produce no output; usually the script defines some new Scheme functions which you can then use in addition the the Scheme and VRR functions. See Section 5.1.10 [The Scheme console], page 38 to find out how to work with the console and Chapter 6 [Scheme], page 46 to learn about VRR Scheme functions.

• I have tried to transform an object, but I got the message "This selection cannot be transformed." What am I doing wrong?

You are trying to transform a dependent object. You might, for example, have snapped one or more anchors of the object to some hangers (not mouse-clicks). If you try to move the object, but not the ones containing the hangers, then moving it would disobey the dependencies.

If you want to free the object from dependencies, rehang its anchors to mouse-clicks (see Section 4.1 [Anchors and hangers], page 27, Section 3.6.3 [Anchor rehang], page 19, Section 3.6 [Snap – introducing geometric dependencies], page 18).

• What is the difference between transformation and anchor rehanging?

Transformations modify the shape of graphic objects, but keep geometric dependencies unchanged. All snapped objects stay snapped in the same way. In contrast to this, rehanging of anchors changes geometric dependencies, removes some of them or creates new. See Section 3.4 [Transformations of graphic objects], page 11, Section 3.6.3 [Anchor rehang], page 19, Section 3.6.1 [What is snap? What is it good for?], page 18.

• While creating a new object, I clicked a menu command and the object disappeared. How come?

By clicking the menu command, you interrupted the operation of the editor (see Section 5.3 [The mechanism of creating new graphic objects], page 39). Almost all commands are incompatible with the creation of a new graphic object, that they cancel the current operation, which deletes the partially created object. The main reason for this is that the editor uses undo history items to enable the "Step back" feature and does not allow any other actions to interfere with its undo items.

• **How do you keep a programmer in the shower all day?**

Give him a bottle of shampoo which says "lather, rinse, repeat."

• **What is the difference between "local" and "global" undo?**

The local undo belongs to a page (each page has its independent undo history) and stores actions performed on the contents of the page. The global undo stores the other actions not connected to contents of any page. See Section 4.3 [Documents and pages], page 29.

• **I clicked "undo" when editing a page, but instead of undoing my last action, it made the page's View disappear. Why?**

You actually clicked the "global undo" which undid the last action not connected to the contents of any page. In your case, it was the creation of the edited page, so you deleted it. Using the "global redo" command, you restore the page with all its contents untouched. See Section 4.3 [Documents and pages], page 29.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W

Y

Z