

---

# **Gaphas Documentation**

***Release 3.0.0-beta.6***

**Arjan J. Molenaar**

**Jan 20, 2021**



## THE BASICS

<b>1</b>	<b>Table of Contents</b>	<b>3</b>
	<b>Index</b>	<b>33</b>



---

**Important:** This documentation is in the process of being updated for Gaphas 3.0.

---

Gaphas is the diagramming widget library for Python.

Gaphas has been built with extensibility in mind. It can be used for many drawing purposes, including vector drawing applications, and diagram drawing tools.

The basic idea is:

- Gaphas has a [Model-View-Controller](#) design.
- A model is presented as a protocol in Gaphas. This means that it's very easy to define a class that acts as a model.
- A model can be visualized by one or more Views.
- A constraint solver is used to maintain item constraints and inter-item constraints.
- The item (and user) should not be bothered with things like bounding-box calculations.
- Very modular: The view contains the basic features. Painters and tools can be swapped out as needed.
- Rendering using [Cairo](#). This implies the diagrams can be exported in a number of formats, including PNG and SVG.

Gaphas is released under the terms of the Apache Software License, version 2.0.

- Git repository: <https://github.com/gaphor/gaphas>
- Python Package index (PyPI): <https://pypi.org/project/gaphas>

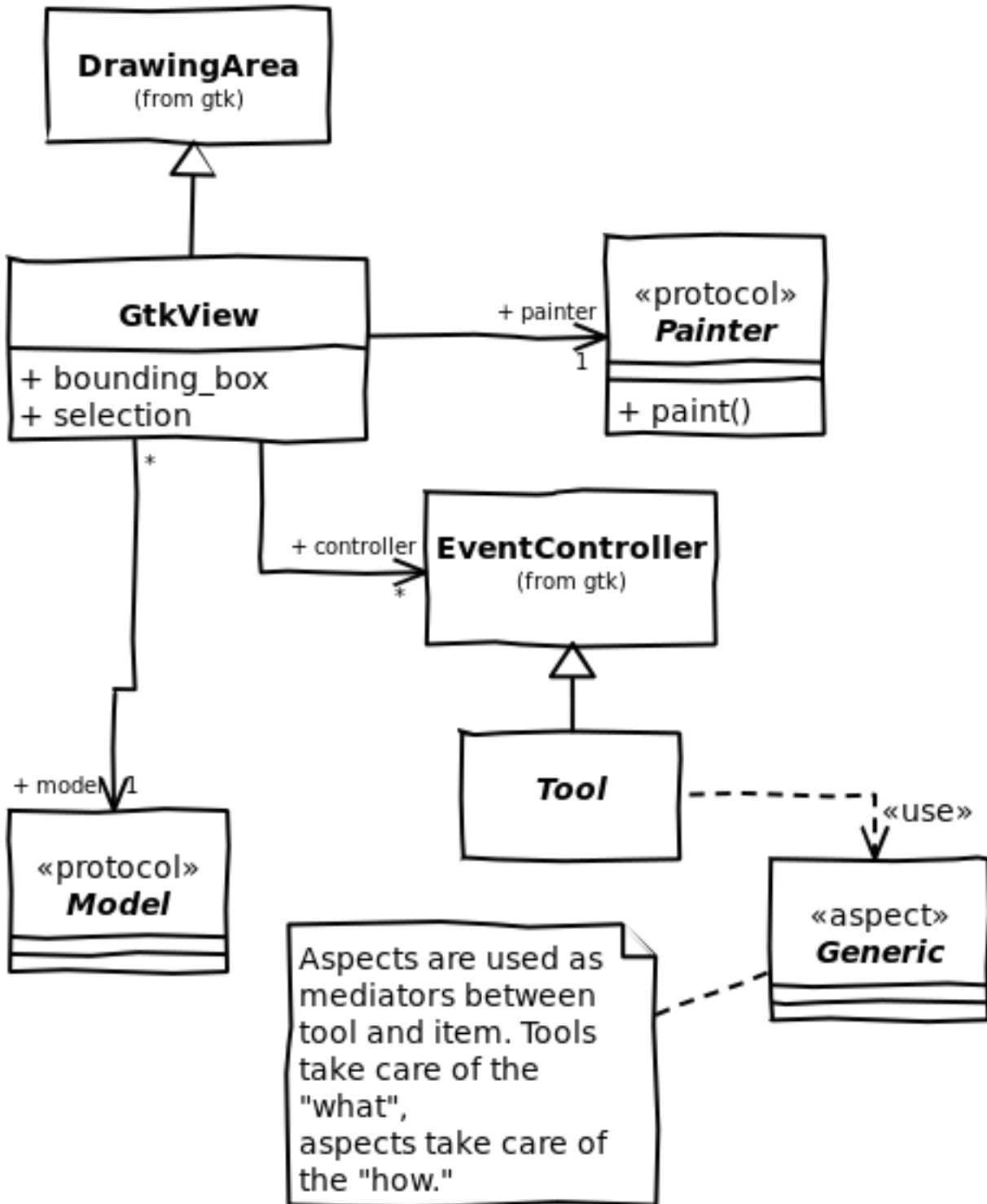


## TABLE OF CONTENTS

### 1.1 Class diagram

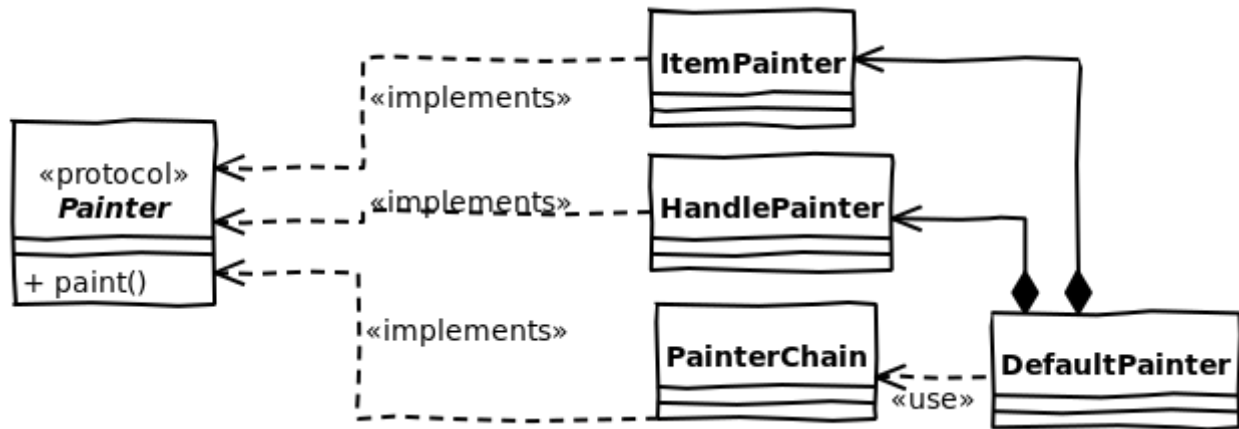
This class diagram describes the basic layout of Gaphas.

The central class is `GtkView`. It takes a model. A default implementation is provided by *`gaphas.Canvas`*. A view is rendered by `Painters`. Interaction is handled by `Tools`.

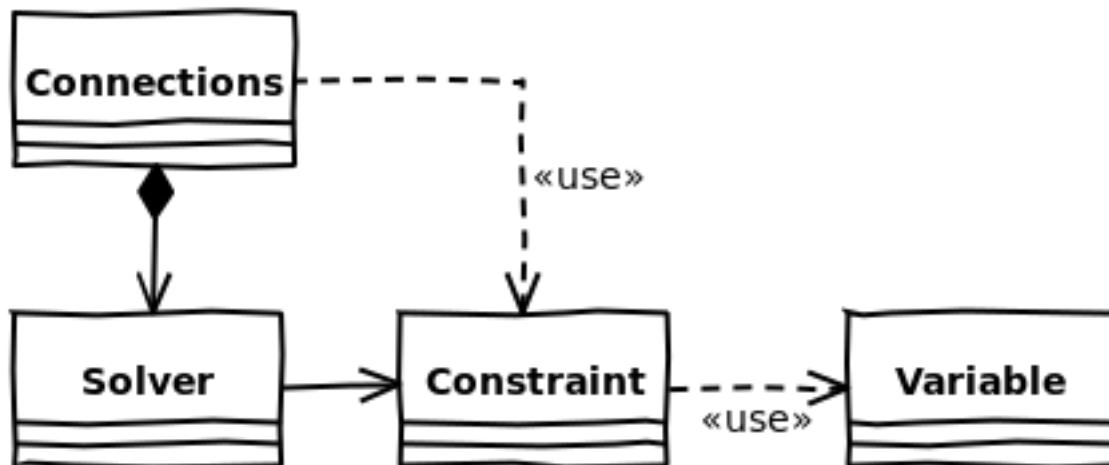


Painting is done by painters. Each painter will paint a layer of the canvas.

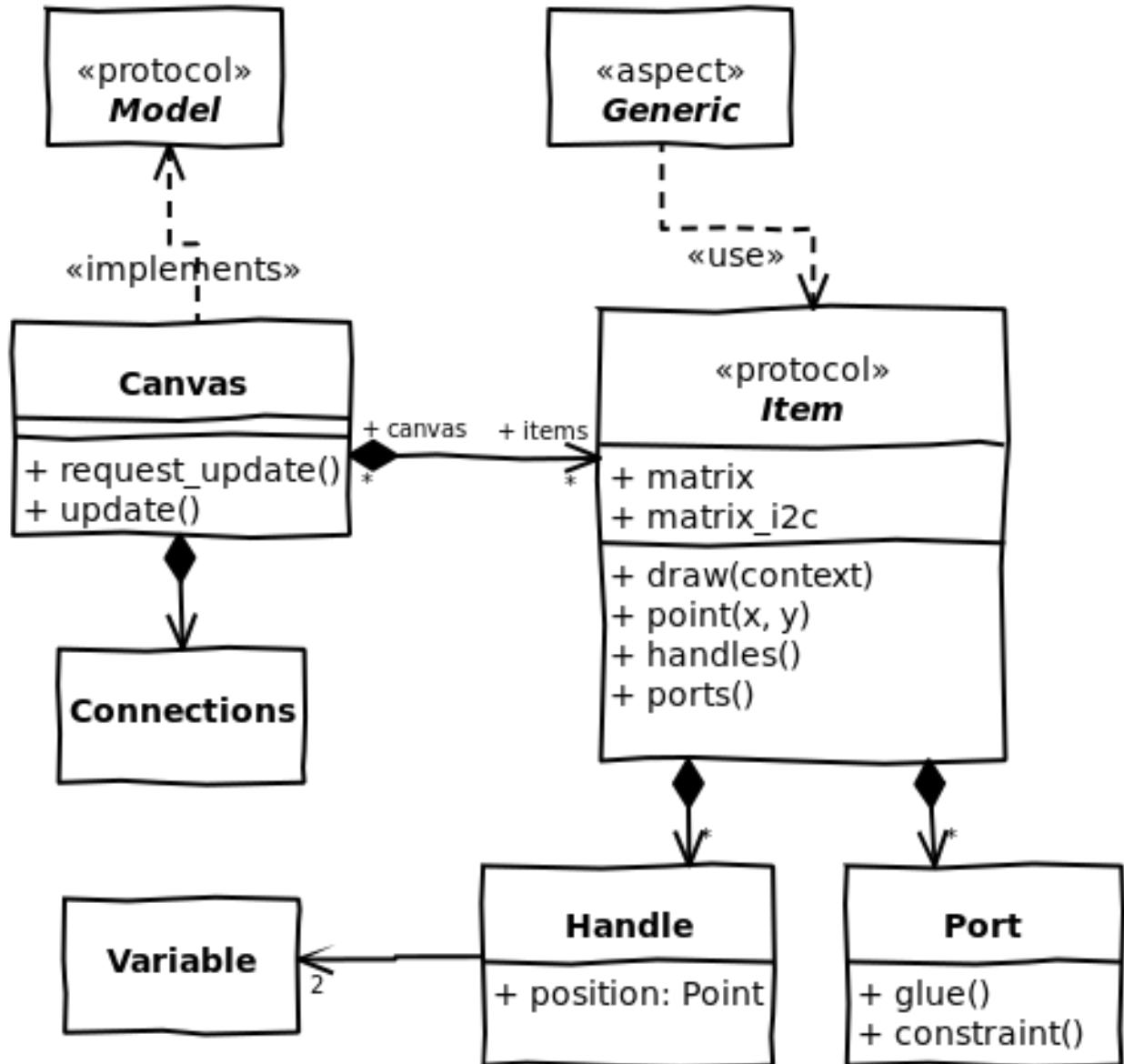




Besides the view, there is constraint based connection management. Constraints can be used within an item, and to connect different items.



A default model and item implementations, a line and an element.



## 1.2 Interacting with diagrams

Tools are used to handle user actions, like moving a mouse pointer over the screen and clicking on items in the canvas.

Tools are registered on the `View`. They have some internal state (e.g. when and where a mouse button was pressed). Therefore tools can not be reused by different views<sup>1</sup>.

Tools are simply `Gtk.EventController` instances. For a certain action to happen multiple user events are used. For example a click is a combination of a button press and button release event (only talking mouse clicks now). In most cases also some movement is done. A sequence of a button press, some movement and a button release is treated as one transaction. Once a button is pressed the tool registers itself as the tool that will deal with all subsequent events (a grab).

Several events can happen based on user events. E.g.:

<sup>1</sup> as opposed to versions < 0.5, where tools could be shared among multiple views.

- item is hovered over (motion)
- item is hovered over while another item is being moved (`press, motion`)
- item is hovered over while dragging something else (DnD; `press, move`)
- grabbed (button press on item)
- handle is grabbed (button press on handle)
- center of line segment is grabbed (will cause segment to split; button press on line)
- ungrabbed (button release)
- move (item is moved -> hover + grabbed)
- key is pressed
- key is released
- modifier is pressed (e.g. may cause mouse pointer to change, giving a hint about what a grab event will do).

There is a lot of behaviour possible and it can depend on the kind of diagrams that are created what has to be done.

To organize the event sequences and keep some order in what the user is doing Tools are used. Tools define what has to happen (find a handle nearly the mouse cursor, move an item).

Gaphas contains a set of default tools. Each tool is meant to deal with a special part of the view. A list of responsibilities is also defined here:

**hover tool** First thing a user wants to know is if the mouse cursor is over an item. The `HoverTool` makes that explicit. - Find a handle or item, if found, mark it as the `hovered_item`

**item tool** Items are the elements that are actually providing any (visual) meaning to the diagram. Item-Tool deals with moving them around. The tool makes sure the right subset of selected elements are moved (e.g. you don't want to move a nested item if its parent item is already moved, this gives funny visual effects)

- On click: find an item, if found become the grabbed tool and set the item as focused. If a used clicked on a handle position that is taken into account
- On motion: move the selected items (only the ones that have no selected parent items)
- On release: release grab and release item

The item tool invokes the *Move* aspect, or the *HandleMove* aspect in case a handle is being grabbed.

**rubberband tool** If no handle or item is selected a rubberband selection is started.

**scroll and zoom tool** Handy tools for moving the canvas around and zooming in and out. Convenience functionality, basically.

There is one more tool, that has not been mentioned yet:

**placement tool** A special tool to use for placing new items on the screen.

As said, tools define *what* has to happen, they don't say how. Take for example finding a handle: on a normal element (a box or something) that would mean find the handle in one of the corners. On a line, however, that may also mean a not-yet existing handle in the middle of a line segment (there is a functionality that splits the line segment).

The *how* is defined by so called aspects<sup>2</sup>.

<sup>2</sup> not the AOP term. The term aspect is coming from a paper by Dirk Riehe: The Tools and Materials metaphore <https://wiki.c2.com/?ToolsAndMaterialsMetaphor..>>.

### 1.2.1 Separating the *What* from the *How*

The *what* is decided in a tool. Based on this the *how* logic can be applied to the item at hand. For example: if an item is clicked, it should be marked as the focused item. Same for dragging: if an item is dragged it should be updated based on the event information. It may even apply this to all other selected items.

The how logic depends actually on the item it is applied to. Lines have different behaviours than boxes for example. In Gaphas this has been resolved by defining a generic methods. To put it simple: a generic method is a factory that returns a specific method (or instance of a class, as we do in gaphas) based on its parameters.

The advantage is that more complex behaviour can be composed. Since the decision on what should happen is done in the tool, the aspect which is then used to work on the item ensures a certain behaviour is performed.

## 1.3 Connections

A Port defines a connectable part of an item. Handles can connect to ports to make connections between items.

### 1.3.1 Constraints

Diagram items can have internal constraints, which can be used to position item's ports within an item itself.

For example, *Element* item could create constraints to position ports over its edges of rectangular area. The result is duplication of constraints as *Element* already constraints position of handles to keep them in a rectangle.

For example, a horizontal line could be implemented like:

```
class HorizontalLine(gaphas.item.Item):
    def __init__(self, connections: gaphas.connections.Connections):
        super(HorizontalLine, self).__init__()

        self.start = Handle()
        self.end = Handle()

        self.port = LinePort(self.start.pos, self.end.pos)

        connections.add_constraint(self,
                                   constraint(horizontal=(self.start.pos, self.end.pos)))
```

### 1.3.2 Connections

Connection between two items is established by creating a constraint between handle's position and port's positions (positions are constraint solver variables).

To create a constraint between two items, the constraint needs a common coordinate space (each item has it's own origin coordinate). This can be done with the *gaphas.position.MatrixProjection* class, which translates coordinates to a common ("canvas") coordinate space where they can be used to connect two different items.

Examples of ports can be found in Gaphas and Gaphor source code

- *gaphas.item.Line* and *gaphas.item.Element* classes
- Gaphor interface and lifeline items have own specific ports

## 1.4 Constraint Solver

Gaphas' constraint solver can be considered the heart of the library. The constraint solver ('solver' for short) is used to manage constraints. Both constraint internal to an item, such as handle alignment for a box, as well as inter-item connections, for example when a line is connected to a box. The solver is called during the update of the canvas.

A solver contains a set of constraints. Each constraint in itself is pretty straightforward (e.g. variable "a" equals variable "b"). Did I say variable? Yes I did. Let's start at the bottom and work our way to the solver.

A `Variable` is a simple class, contains a value. It behaves like a `float` in many ways. There is one typical thing about `Variables`: they can be added to `Constraints`.

`Constraint` are basically equations. The trick is to make all constraints true. That can be pretty tricky, since a `Variable` can play a role in more than one `Constraint`. Constraint solving is overseen by the `Solver` (ah, there it is).

`Constraints` are instances of `Constraint` class. More specific: subclasses of the `Constraint` class. A `Constraint` can perform a specific trick, e.g. centre one `Variable` between two other `Variables` or make one `Variable` equal to another `Variable`.

It's the `Solver`'s job to make sure all constraint are true in the end. In some cases this means a constraint needs to be resolved twice, but the `Solver` sees to it that no deadlocks occur.

### 1.4.1 Variables

When a variable is assigned a value it marks itself `__dirty__`. As a result it will be resolved the next time the solver is asked to.

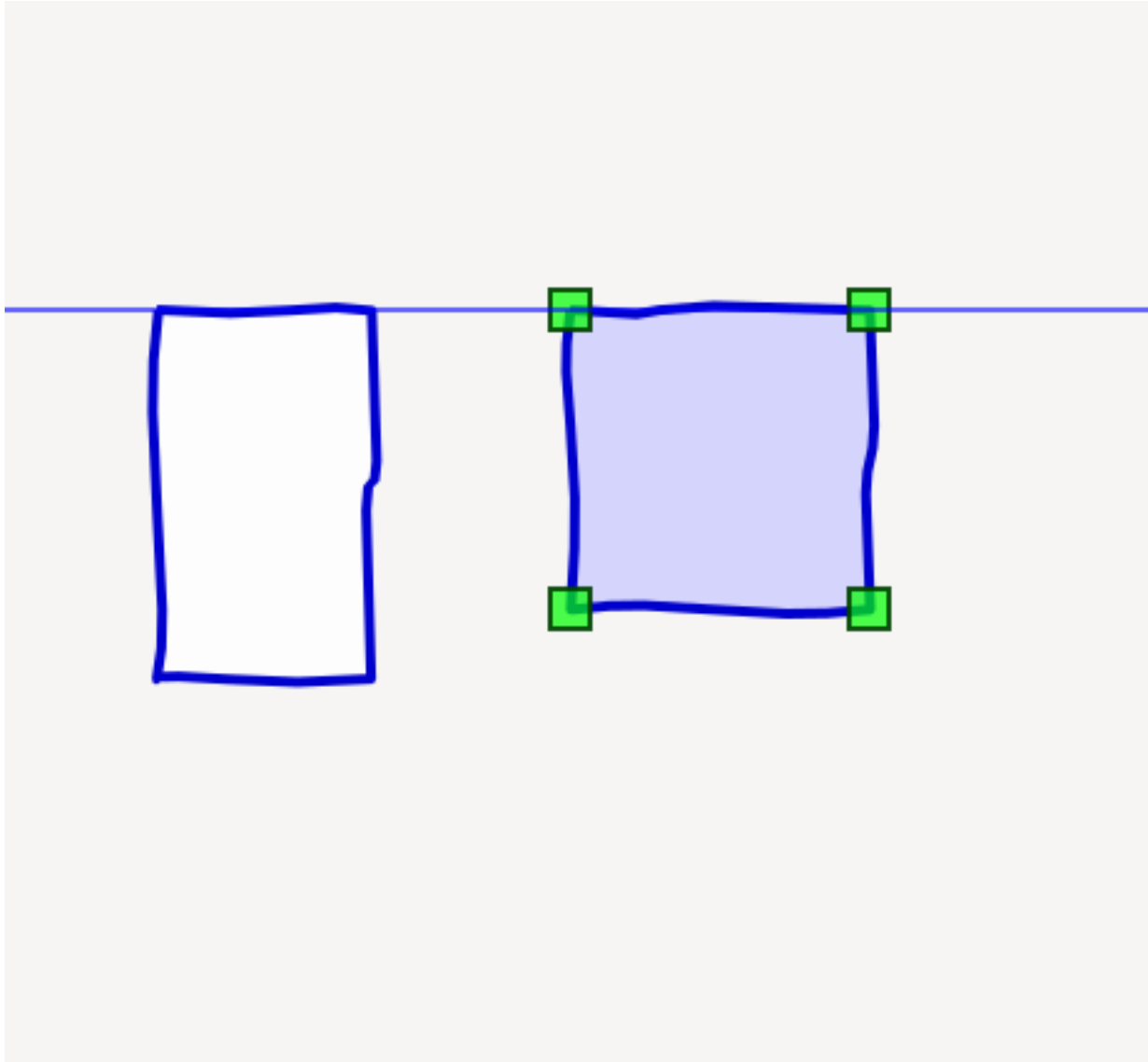
Each variable has a specific "strength". Strong variables can not be changed by weak variables, but weak variables can change when a new value is assigned to a stronger variable. The `Solver` always tries to solve a constraint for the weakest variable. If two variables have equal strength, however, the variable that is most recently changed is considered slightly stronger than the not (or earlier) changed variable.

---

The `Solver` can be found at: <https://github.com/gaphor/gaphas/blob/master/gaphas/solver/>, along with `Variable` and the `Constraint` base class.

## 1.5 Guides

Guides are a tool to align elements with one another.



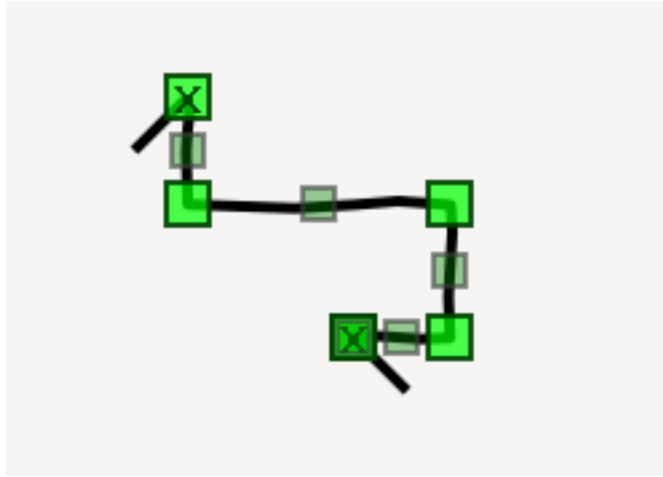
Guides consist of a couple of elements: aspects that hook into the item-drag cycle, and a dedicated painter.

```
>>> from gaphas.view import GtkView
>>> from gaphas.painter import PainterChain, ItemPainter, HandlePainter
>>> from gaphas.tool import item_tool, scroll_tool, zoom_tool
>>> from gaphas.guide import GuidePainter
>>> view = GtkView()
>>> view.painter = (
...     PainterChain()
...     .append(ItemPainter(view.selection))
...     .append(HandlePainter(view))
...     .append(GuidePainter(view))
... )
>>> view.add_controller(item_tool(view))
>>> view.add_controller(scroll_tool(view))
>>> view.add_controller(zoom_tool(view))
```

You need to hook up the `GuidePainter`. The aspect are loaded as soon as the module is loaded.

## 1.6 Line Segments

The line segment functionality is an add-on, that will allow the user to add line segments to a line, and merge them.



To use this behavior, import the `gaphas.segment` module and add `LineSegmentPainter` to the list of painters for the view. Add `segment_tool` to the view as a controller, in order to activate the split/merge segment behavior for `Line` instances.

```
>>> from gaphas.view import GtkView
>>> from gaphas.painter import PainterChain, ItemPainter, HandlePainter
>>> from gaphas.tool import item_tool, scroll_tool, zoom_tool
>>> from gaphas.segment import LineSegmentPainter, segment_tool
>>> view = GtkView()
>>> view.painter = (
...     PainterChain()
...     .append(ItemPainter(view.selection))
...     .append(HandlePainter(view))
...     .append(LineSegmentPainter(view.selection))
... )
>>> view.add_controller(segment_tool(view))
>>> view.add_controller(item_tool(view))
>>> view.add_controller(scroll_tool(view))
>>> view.add_controller(zoom_tool(view))
```

## 1.7 API reference

### 1.7.1 View

View is the central class in Gaphas. It shows your diagram and allows you to interact with it.

**class** `gaphas.view(GtkView(*args: Any, **kwargs: Any))`

GTK+ widget for rendering a `gaphas.view.model.Model` to a screen. The view uses `Tools` to handle events and `Painters` to draw. Both are configurable.

The widget already contains adjustment objects (*hadjustment*, *vadjustment*) to be used for scrollbars.

This view registers itself on the model, so it will receive update events.

**property matrix**

Model root to view transformation matrix.

**get\_matrix\_i2v** (*item*: [gaphas.item.Item](#)) → [gaphas.matrix.Matrix](#)

Get Item to View matrix for *item*.

**get\_matrix\_v2i** (*item*: [gaphas.item.Item](#)) → [gaphas.matrix.Matrix](#)

Get View to Item matrix for *item*.

**property model**

The model.

**property painter**

Painter for drawing the view.

**property bounding\_boxPainter**

Special painter for calculating item bounding boxes.

**property selection**

Selected, focused and hovered items.

**property bounding\_box**

The bounding box of the complete view, relative to the view port.

**property hadjustment**

Gtk adjustment object for use with a scrollbar.

**property vadjustment**

Gtk adjustment object for use with a scrollbar.

**add\_controller** (\**controllers*: [gi.repository.Gtk.EventController](#)) → None

Add a controller.

A convenience method, so you have a place to store the event controllers. Events controllers are linked to a widget (in GTK3) on creation time, so calling this method is not necessary.

**remove\_controller** (*controller*: [gi.repository.Gtk.EventController](#)) → bool

Remove a controller.

The event controller's propagation phase is set to [Gtk.PropagationPhase.NONE](#) to ensure it's not invoked anymore.

**NB. The controller is only really removed from the widget when it's destroyed!** This is a Gtk3 limitation.

**remove\_all\_controllers** () → None

Remove all registered controllers.

**zoom** (*factor*: *float*) → None

Zoom in/out by factor *factor*.

**get\_items\_in\_rectangle** (*rect*: [Tuple\[float, float, float, float\]](#), *contain*: *bool* = *False*) → [Iterable\[gaphas.item.Item\]](#)

Return the items in the rectangle 'rect'.

Items are automatically sorted in model's processing order.

**get\_item\_bounding\_box** (*item*: [gaphas.item.Item](#)) → [gaphas.geometry.Rectangle](#)

Get the bounding box for the item, in view coordinates.

**queue\_redraw** () → None

Redraw the entire view.



**request\_update** (*items*: Iterable[gaphas.item.Item], *matrix\_only\_items*: Iterable[gaphas.item.Item] = (), *removed\_items*: Iterable[gaphas.item.Item] = ()) → None

Request update for items.

Items will get a full update treatment, while *matrix\_only\_items* will only have their bounding box recalculated.

**all\_dirty\_matrix\_items** () → Set[gaphas.item.Item]

Recalculate matrices of the items. Items' children matrices are recalculated, too.

Return items, which matrices were recalculated.

**update\_bounding\_box** (*items*: Collection[gaphas.item.Item]) → None

Update the bounding boxes of the model items for this view, in model coordinates.

## 1.7.2 Model

### Protocols

Although `gaphas.Canvas` can be used as a default model, any class that adhere's to the Model protocol can be used as a model.

**class** `gaphas.view.model.Model` (\*args, \*\*kws)

Any class that adhere's to the Model protocol can be used as a model for `GtkView`.

#### property connections

The connections instance used for this model.

**get\_all\_items** () → Iterable[gaphas.item.Item]

Iterate over all items in the order they need to be rendered in.

Normally that will be depth-first.

**get\_parent** (*item*: gaphas.item.Item) → Optional[gaphas.item.Item]

Get the parent item of an item.

Returns None if there is no parent item.

**get\_children** (*item*: Optional[gaphas.item.Item]) → Iterable[gaphas.item.Item]

Iterate all direct child items of an item.

**sort** (*items*: Collection[gaphas.item.Item]) → Iterable[gaphas.item.Item]

Sort a collection of items in the order they need to be rendered in.

**request\_update** (*item*: gaphas.item.Item, *update*: bool = True, *matrix*: bool = True) → None

Request update for an item.

#### Parameters

- **item** (Item) – The item to be updated
- **update** (bool) – True if it needs a full update (incl. bounding box calculation)
- **matrix** (bool) – True if only the matrix has been updated (item has moved/resized)

**update\_now** (*dirty\_items*: Collection[gaphas.item.Item], *dirty\_matrix\_items*: Collection[gaphas.item.Item]) → None

This method is called during the update process.

It will allow the model to do some additional updating of it's own.

**register\_view** (*view: gaphas.view.model.View*) → None

Allow a view to be registered.

Registered views should receive update requests for modified items.

**unregister\_view** (*view: gaphas.view.model.View*) → None

Unregister a previously registered view.

If a view is not registered, nothing should happen.

An item should implement these methods, so it can be rendered by the View. Not that painters or tools can require additional methods.

**class** `gaphas.item.Item(*args, **kwargs)`

This protocol should be implemented by model items.

All items that are rendered on a view.

**property matrix**

The “local”, item-to-parent matrix.

**property matrix\_i2c**

Matrix from item to toplevel.

**handles** () → Sequence[*gaphas.connector.Handle*]

Return a list of handles owned by the item.

**ports** () → Sequence[*gaphas.connector.Port*]

Return list of ports owned by the item.

**point** (*x: float, y: float*) → float

Get the distance from a point (x, y) to the item.

x and y are in item coordinates.

A distance of 0 means the point is on the item.

**draw** (*context: gaphas.item.DrawContext*) → None

Render the item to a canvas view. Context contains the following attributes:

- *cairo*: the CairoContext use this one to draw
- *selected, focused, hovered*: view state of items (True/False)

## Default implementations

### Canvas

The default implementation for a Model, is a class called Canvas.

**class** `gaphas.canvas.Canvas`

Container class for items.

**add** (*item, parent=None, index=None*)

Add an item to the canvas.

```
>>> c = Canvas()
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> len(c._tree.nodes)
1
```

(continues on next page)

(continued from previous page)

```
>>> i._canvas is c
True
```

**remove (item)**

Remove item from the canvas.

```
>>> c = Canvas()
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> c.remove(i)
>>> c._tree.nodes
[]
>>> i._canvas
```

**reparent (item, parent, index=None)**

Set new parent for an item.

**get\_all\_items ()** → Iterable[Item]

Get a list of all items.

```
>>> c = Canvas()
>>> c.get_all_items()
[]
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> c.get_all_items()
[<gaphas.item.Item ...>]
```

**get\_root\_items ()**

Return the root items of the canvas.

```
>>> c = Canvas()
>>> c.get_all_items()
[]
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> ii = item.Item()
>>> c.add(ii, i)
>>> c.get_root_items()
[<gaphas.item.Item ...>]
```

**get\_parent (item: Item)** → Optional[Item]See *tree.Tree.get\_parent()*.

```
>>> c = Canvas()
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> ii = item.Item()
>>> c.add(ii, i)
>>> c.get_parent(i)
>>> c.get_parent(ii)
<gaphas.item.Item ...>
```

**get\_children** (*item: Optional[Item]*) → Iterable[Item]  
 See *tree.Tree.get\_children()*.

```
>>> c = Canvas()
>>> from gaphas import item
>>> i = item.Item()
>>> c.add(i)
>>> ii = item.Item()
>>> c.add(ii, i)
>>> iii = item.Item()
>>> c.add(iii, ii)
>>> list(c.get_children(iii))
[]
>>> list(c.get_children(ii))
[<gaphas.item.Item ...>]
>>> list(c.get_children(i))
[<gaphas.item.Item ...>]
```

**sort** (*items: Iterable[Item]*) → Iterable[Item]  
 Sort a list of items in the order in which they are traversed in the canvas (Depth first).

```
>>> c = Canvas()
>>> from gaphas import item
>>> i1 = item.Line()
>>> c.add(i1)
>>> i2 = item.Line()
>>> c.add(i2)
>>> i3 = item.Line()
>>> c.add(i3)
>>> c.update_now((i1, i2, i3)) # ensure items are indexed
>>> s = c.sort([i2, i3, i1])
>>> s[0] is i1 and s[1] is i2 and s[2] is i3
True
```

**get\_matrix\_i2c** (*item: Item*) → matrix.Matrix  
 Get the Item to Canvas matrix for *item*.

**item:** The item who's item-to-canvas transformation matrix should be found

**calculate:** True will allow this function to actually calculate it, instead of raising an *AttributeError* when no matrix is present yet. Note that out-of-date matrices are not recalculated.

**request\_update** (*item: Item, update: bool = True, matrix: bool = True*) → None  
 Set an update request for the item.

```
>>> c = Canvas()
>>> from gaphas import item
>>> i = item.Item()
>>> ii = item.Item()
>>> c.add(i)
>>> c.add(ii, i)
>>> len(c._dirty_items)
0
>>> c.update_now((i, ii))
>>> len(c._dirty_items)
0
```

**request\_matrix\_update** (*item*)  
 Schedule only the matrix to be updated.

**update\_now** (*\*\*kwargs*)

Decorate function with a mutex that prohibits recursive execution.

**register\_view** (*view: View*) → None

Register a view on this canvas.

This method is called when setting a canvas on a view and should not be called directly from user code.

**unregister\_view** (*view: View*) → None

Unregister a view on this canvas.

This method is called when setting a canvas on a view and should not be called directly from user code.

## Items

Gaphas provides two default items, an box-like element and a line shape.

**class** `gaphas.item.Element` (*connections: Connections, width: float = 10, height: float = 10, \*\*kwargs: object*)

An Element has 4 handles (for a start):

NW +—+ NE || SW +—+ SE

**property width**

Width of the box, calculated as the distance from the left and right handle.

**property height**

Height.

**handles** () → Sequence[`gaphas.connector.Handle`]

Return a list of handles owned by the item.

**ports** () → Sequence[`gaphas.connector.Port`]

Return list of ports.

**point** (*x: float, y: float*) → float

Distance from the point (x, y) to the item.

```
>>> e = Element()
>>> e.point(20, 10)
10.0
```

**class** `gaphas.item.Line` (*connections: Connections, \*\*kwargs: object*)

A Line item.

### Properties:

- **fuzziness (0.0..n): an extra margin that should be taken into** account when calculating the distance from the line (using `point()`).
- **orthogonal (bool): whether or not the line should be** orthogonal (only straight angles)
- **horizontal:** first line segment is horizontal
- **line\_width:** width of the line to be drawn

This line also supports arrow heads on both the begin and end of the line. These are drawn with the methods `draw_head(context)` and `draw_tail(context)`. The coordinate system is altered so the methods do not have to know about the angle of the line segment (e.g. drawing a line from (10, 10) via (0, 0) to (10, -10) will draw an arrow point).

**update\_orthogonal\_constraints** (*orthogonal: bool*) → None

Update the constraints required to maintain the orthogonal line.

The actual constraints attribute (`_orthogonal_constraints`) is observed, so the undo system will update the contents properly

**opposite** (*handle: gaphas.connector.Handle*) → *gaphas.connector.Handle*

Given the handle of one end of the line, return the other end.

**handles** () → Sequence[*gaphas.connector.Handle*]

Return a list of handles owned by the item.

**ports** () → Sequence[*gaphas.connector.Port*]

Return list of ports.

**point** (*x: float, y: float*) → float

```
>>> a = Line()
>>> a.handles()[1].pos = 25, 5
>>> a._handles.append(a._create_handle((30, 30)))
>>> a.point(-1, 0)
1.0
>>> f"{a.point(5, 4):.3f}"
'2.942'
>>> f"{a.point(29, 29):.3f}"
'0.784'
```

**draw\_head** (*context: gaphas.item.DrawContext*) → None

Default head drawer: move cursor to the first handle.

**draw\_tail** (*context: gaphas.item.DrawContext*) → None

Default tail drawer: draw line to the last handle.

**draw** (*context: gaphas.item.DrawContext*) → None

Draw the line itself.

See `Item.draw(context)`.

## 1.7.3 Painters

Painters are used to draw the view.

### Protocols

Each painter adheres to the `Painter` protocol.

**class** `gaphas.painter.Painter` (\*args, \*\*kws)

Painter interface.

**paint** (*items: Collection[gaphas.item.Item], cairo: gaphas.types.CairoContext*) → Optional[Dict[*gaphas.item.Item, gaphas.geometry.Rectangle*]]

Do the paint action (called from the View).

Some painters, such as `FreeHandPainter` and `BoundingBoxPainter`, require a special painter protocol:

**class** `gaphas.painter.painter.ItemPainterType` (\*args, \*\*kws)

**paint\_item** (*item*: [gaphas.item.Item](#), *cairo*: [gaphas.types.CairoContext](#)) → None  
 Draw a single item.

**paint** (*items*: [Collection\[gaphas.item.Item\]](#), *cairo*: [gaphas.types.CairoContext](#)) → None  
 Do the paint action (called from the View).

## Default implementations

**class** [gaphas.painter.PainterChain](#)  
 Chain up a set of painters.

**append** (*painter*: [gaphas.painter.painter.Painter](#)) → [gaphas.painter.chain.PainterChain](#)  
 Add a painter to the list of painters.

**prepend** (*painter*: [gaphas.painter.painter.Painter](#)) → [gaphas.painter.chain.PainterChain](#)  
 Add a painter to the beginning of the list of painters.

**class** [gaphas.painter.ItemPainter](#) (*selection*: [Optional\[gaphas.view.selection.Selection\]](#) = None)

**class** [gaphas.painter.HandlePainter](#) (*view*: [GtkView](#))  
 Draw handles of items that are marked as selected in the view.

**class** [gaphas.painter.BoundingBoxPainter](#) (*item\_painter*: [gaphas.painter.painter.ItemPainterType](#))  
 This specific case of an ItemPainter is used to calculate the bounding boxes (in cairo device coordinates) for the items.

**class** [gaphas.painter.FreeHandPainter](#) (*subpainter*: [gaphas.painter.painter.ItemPainterType](#), *sloppiness*: [float](#) = 0.5)

This painter is a wrapper for an Item painter. The Cairo context is modified to allow for a sloppy, hand written drawing style.

Range [0..2.0] gives acceptable results.

- Draftsman: 0.0
- Artist: 0.25
- Cartoonist: 0.5
- Child: 1.0
- Drunk: 2.0

## Rubberband tool

A special painter is used to display rubberband selection. This painter shares some state with the rubberband tool.

**class** [gaphas.tool.rubberband.RubberbandPainter](#) (*rubberband\_state*: [gaphas.tool.rubberband.RubberbandState](#))

The rubberband painter should be used in conjunction with the rubberband tool.

[RubberbandState](#) should be shared between the two.

## 1.7.4 Tools

Tools are used to interact with the view.

Each tool is basically a function that produces a `Gtk.EventController`. The event controllers are already configured.

`gaphas.tool.hover_tool` (*view*: `gaphas.view.gtkview.GtkView`) → `gi.repository.Gtk.EventController`

Highlight the currently hovered item.

`gaphas.tool.item_tool` (*view*: `gaphas.view.gtkview.GtkView`) → `gi.repository.Gtk.GestureDrag`

Handle item movement and movement of handles.

`gaphas.tool.placement_tool` (*view*: `gaphas.view.gtkview.GtkView`, *factory*: `Callable[[gaphas.item.Item], gi.repository.Gtk.GestureDrag]`, *handle\_index*: `int`) → `gi.repository.Gtk.GestureDrag`

Place a new item on the model.

`gaphas.tool.rubberband_tool` (*view*, *rubberband\_state*)

Rubberband selection tool.

Should be used in conjunction with `RubberbandPainter`.

`gaphas.tool.scroll_tool` (*view*: `gaphas.view.gtkview.GtkView`, *speed*: `int = 5`) → `gi.repository.Gtk.EventControllerScroll`

Scroll tool recognized 2 finger scroll gestures.

`gaphas.tool.zoom_tool` (*view*: `gaphas.view.gtkview.GtkView`) → `gi.repository.Gtk.GestureZoom`

Create a zoom tool as a `Gtk.Gesture`.

Note: we need to keep a reference to this gesture, or else it will be destroyed.

The central part for Gaphas is the View. That's the class that ensures stuff is displayed and can be interacted with.

## 1.7.5 Handles and Ports

To connect one item to another, you need something to connect, and something to connect to. These roles are fulfilled by `Handle` and `Port`.

The `Handle` is an item you normally see on screen as a small square, either green or red. Although the actual shape depends on the `Painter` used.

Ports represent the receiving side. A port decides if it wants a connection with a handle. If it does, a constraint can be created and this constraint will be managed by a `Connections` instance. It is not uncommon to create special ports to suite your application's behavior, whereas `Handles` are rarely subtyped.

### Handle

```
class gaphas.connector.Handle (pos: Tuple[float, float] = (0, 0), strength: int = 20, connectable:  
                                bool = False, movable: bool = True)
```

`Handles` are used to support modifications of `Items`.

If the handle is connected to an item, the `connected_to` property should refer to the item. A `disconnect` handler should be provided that handles all disconnect behaviour (e.g. clean up constraints and `connected_to`).

Note for those of you that use the `Pickle` module to persist a canvas: The property `disconnect` should contain a callable object (with `__call__()` method), so the pickle handler can also pickle that. `Pickle` is not capable of pickling `instancemethod` or `function` objects.



**property pos**

The Handle's position

**property connectable**

Can this handle actually connect to a port?

**property movable**

Can this handle be moved by a mouse pointer?

**property visible**

Is this handle visible to the user?

**Port**

The Port class. There are two default implementations: `LinePort` and `PointPort`.

**class** `gaphas.connector.Port`

Port connectable part of an item.

The Item's handle connects to a port.

**glue** (*pos*: `Tuple[SupportsFloat, SupportsFloat]`) → `Tuple[Tuple[float, float], float]`

Get glue point on the port and distance to the port.

**constraint** (*item*: `Item`, *handle*: `Handle`, *glue\_item*: `Item`) → `Constraint`

Create connection constraint between item's handle and glue item.

**class** `gaphas.connector.LinePort` (*start*: `gaphas.position.Position`, *end*: `gaphas.position.Position`)

Port defined as a line between two handles.

**glue** (*pos*: `Tuple[SupportsFloat, SupportsFloat]`) → `Tuple[Tuple[float, float], float]`

Get glue point on the port and distance to the port.

```
>>> p1, p2 = (0.0, 0.0), (100.0, 100.0)
>>> port = LinePort(p1, p2)
>>> port.glue((50, 50))
((50.0, 50.0), 0.0)
>>> port.glue((0, 10))
((5.0, 5.0), 7.0710678118654755)
```

**constraint** (*item*: `Item`, *handle*: `Handle`, *glue\_item*: `Item`) → `Constraint`

Create connection line constraint between item's handle and the port.

**class** `gaphas.connector.PointPort` (*point*: `gaphas.position.Position`)

Port defined as a point.

**glue** (*pos*: `Tuple[SupportsFloat, SupportsFloat]`) → `Tuple[Tuple[float, float], float]`

Get glue point on the port and distance to the port.

```
>>> h = Handle((10, 10))
>>> port = PointPort(h.pos)
>>> port.glue((10, 0))
(<Position object on (10, 10)>, 10.0)
```

**constraint** (*item*: `Item`, *handle*: `Handle`, *glue\_item*: `Item`) → `MultiConstraint`

Return connection position constraint between item's handle and the port.

## 1.7.6 Connections

The `Connections` class can be used to manage any type of constraint within, and between items.

**class** `gaphas.connections.Connections` (*solver: Optional[gaphas.solver.solver.Solver] = None*)  
Manage connections and constraints.

**add\_handler** (*handler*)

Add a callback handler.

Handlers are triggered when a constraint has been solved.

**remove\_handler** (*handler*)

Remove a previously assigned handler.

**property solver**

The solver used by this connections instance.

**solve** () → None

Solve all constraints.

**add\_constraint** (*item: gaphas.item.Item, constraint: gaphas.solver.constraint.Constraint*) → `gaphas.solver.constraint.Constraint`

Add a “simple” constraint for an item.

**remove\_constraint** (*item: gaphas.item.Item, constraint: gaphas.solver.constraint.Constraint*) → None

Remove an item specific constraint.

**connect\_item** (*item: gaphas.item.Item, handle: gaphas.connector.Handle, connected: gaphas.item.Item, port: Optional[gaphas.connector.Port], constraint: Optional[gaphas.solver.constraint.Constraint] = None, callback: Optional[Callable[, None]] = None*) → None

Create a connection between two items. The connection is registered and the constraint is added to the constraint solver.

The pair (*item*, *handle*) should be unique and not yet connected.

The callback is invoked when the connection is broken.

### Parameters

- **item** (`Item`) – Connecting item (i.e. a line).
- **handle** (`Handle`) – Handle of connecting item.
- **connected** (`Item`) – Connected item (i.e. a box).
- **port** (`Port`) – Port of connected item.
- **constraint** (`Constraint`) – Constraint to keep the connection in place.
- **callback** (`() -> None`) – Function to be called on disconnection.

`ConnectionError` is raised in case *handle* is already registered on a connection.

**disconnect\_item** (*item: gaphas.item.Item, handle: Optional[gaphas.connector.Handle] = None*) → None

Disconnect the connections of an item.

If *handle* is not None, only the connection for that handle is disconnected.

**remove\_connections\_to\_item** (*item: gaphas.item.Item*) → None

Remove all connections (handles connected to and constraints) for a specific item (to and from the item).

This is some brute force cleanup (e.g. if constraints are referenced by items, those references are not cleaned up).

**reconnect\_item** (*item*: `gaphas.item.Item`, *handle*: `gaphas.connector.Handle`,  
*port*: `Optional[gaphas.connector.Port]` = `None`, *constraint*: `Optional[gaphas.solver.constraint.Constraint]` = `None`) → `None`

Update an existing connection.

This is used to provide a new constraint to the connection. *item* and *handle* are the keys to the to-be-updated connection.

**get\_connection** (*handle*: `gaphas.connector.Handle`) → `Optional[gaphas.connections.Connection]`  
 Get connection information for specified handle.

```
>>> c = Connections()
>>> from gaphas.item import Line
>>> line = Line()
>>> from gaphas import item
>>> i = item.Line()
>>> ii = item.Line()
>>> c.connect_item(i, i.handles()[0], ii, ii.ports()[0])
>>> c.get_connection(i.handles()[0])
Connection(item=<gaphas.item.Line object at 0x...>)
>>> c.get_connection(i.handles()[1])
>>> c.get_connection(ii.handles()[0])
```

**get\_connections** (*item*: `Optional[gaphas.item.Item]` = `None`, *handle*: `Optional[gaphas.connector.Handle]` = `None`, *connected*: `Optional[gaphas.item.Item]` = `None`, *port*: `Optional[gaphas.connector.Port]` = `None`) → `Iterator[gaphas.connections.Connection]`

Return an iterator of connection information.

The list contains (item, handle). As a result an item may be in the list more than once (depending on the number of handles that are connected). If *item* is connected to itself it will also appear in the list.

```
>>> c = Connections()
>>> from gaphas import item
>>> i = item.Line()
>>> ii = item.Line()
>>> iii = item.Line()
>>> c.connect_item(i, i.handles()[0], ii, ii.ports()[0], None)
```

```
>>> list(c.get_connections(item=i))
[Connection(item=<gaphas.item.Line object at 0x...>)]
>>> list(c.get_connections(connected=i))
[]
>>> list(c.get_connections(connected=ii))
[Connection(item=<gaphas.item.Line object at 0x...>)]
```

```
>>> c.connect_item(ii, ii.handles()[0], iii, iii.ports()[0], None)
>>> list(c.get_connections(item=ii))
[Connection(item=<gaphas.item.Line object at 0x...>)]
>>> list(c.get_connections(connected=iii))
[Connection(item=<gaphas.item.Line object at 0x...>)]
```

### 1.7.7 Variables and Position

The most basic class for a solvable value is `Variable`. It acts a lot like a `float`, which makes it easy to work with. Next to that there's `Position`, which is a coordinate  $(x, y)$  defined by two variables.

To support connections between variables, a `MatrixProjection` class is available. It translates a position to a common coordinate space, based on `Item.matrix_i2c`. Normally, it's only `Ports` that deal with item-to-common translation of positions.

**class** `gaphas.solver.Variable` (*value: SupportsFloat = 0.0, strength: int = 20*)

Representation of a variable in the constraint solver.

Each `Variable` has a `value` and a `strength`. In a constraint the weakest variables are changed.

You can even do some calculating with it. The `Variable` always represents a float variable.

The `variable` decorator can be used to easily define variables in classes.

**add\_handler** (*handler: Callable[[gaphas.solver.variable.Variable, float], None]*)  $\rightarrow$  None

Add a handler, to be invoked when the value changes.

**remove\_handler** (*handler: Callable[[gaphas.solver.variable.Variable, float], None]*)  $\rightarrow$  None

Remove a handler.

**notify** (*old: float*)  $\rightarrow$  None

Notify all handlers.

**property strength**

Strength.

**dirty** ()  $\rightarrow$  None

Mark the variable dirty in all attached constraints.

Variables are marked dirty also during constraints solving to solve all dependent constraints, i.e. two equals constraints between 3 variables.

Variables can have different strengths. The higher the number, the stronger the variable. Variables can be `VERY_WEAK` (0), up to `REQUIRED` (100). Other constants are `WEAK` (10) `NORMAL` (20) `STRONG` (30), and `VERY_STRONG` (40).

`gaphas.solver.variable` (*strength=20, varname=None*)

Easy-to-use drop `Variable` descriptor.

```
>>> class A(object):
...     x = variable(varname='_v_x')
...     y = variable(STRONG)
...     def __init__(self):
...         self.x = 12
>>> a = A()
>>> a.x
Variable(12, 20)
>>> a._v_x
Variable(12, 20)
>>> a.x = 3
>>> a.x
Variable(3, 20)
>>> a.y
Variable(0, 30)
```

**class** `gaphas.position.Position` (*x, y, strength=20*)

A point constructed of two `Variable`'s.

```
>>> vp = Position(3, 5)
>>> vp.x, vp.y
(Variable(3, 20), Variable(5, 20))
>>> vp.pos
(Variable(3, 20), Variable(5, 20))
>>> vp[0], vp[1]
(Variable(3, 20), Variable(5, 20))
```

```
class gaphas.position.MatrixProjection (pos: gaphas.position.Position, matrix:
gaphas.matrix.Matrix)
```

One of Gaphas' USP is it's the way it handles connections and the constraint solver.

### 1.7.8 Matrix

The `Matrix` class used to records item placement (translation), scale and skew.

```
class gaphas.matrix.Matrix (xx: float = 1.0, yx: float = 0.0, xy: float = 0.0, yy: float = 1.0, x0: float
= 0.0, y0: float = 0.0, matrix: Optional[cairo.Matrix] = None)
```

Matrix wrapper.

```
>>> Matrix()
Matrix(1.0, 0.0, 0.0, 1.0, 0.0, 0.0)
```

### 1.7.9 Rectangle

```
class gaphas.geometry.Rectangle (x: float = 0, y: float = 0, width: Optional[float] = None, height:
Optional[float] = None, x1: float = 0, y1: float = 0)
```

Python Rectangle implementation. Rectangles can be added (union), substituted (intersection) and points and rectangles can be tested to be in the rectangle.

```
>>> r1= Rectangle(1,1,5,5)
>>> r2 = Rectangle(3,3,6,7)
```

Test if two rectangles intersect:

```
>>> if r1 - r2: 'yes'
'yes'
```

```
>>> r1, r2 = Rectangle(1,2,3,4), Rectangle(1,2,3,4)
>>> r1 == r2
True
```

```
>>> r = Rectangle(-5, 3, 10, 8)
>>> r.width = 2
>>> r
Rectangle(-5, 3, 2, 8)
```

```
>>> r = Rectangle(-5, 3, 10, 8)
>>> r.height = 2
>>> r
Rectangle(-5, 3, 10, 2)
```

**expand** (delta: float) → None

```
>>> r = Rectangle(-5, 3, 10, 8)
>>> r.expand(5)
>>> r
Rectangle(-10, -2, 20, 18)
```

**tuple()** → Tuple[float, float, float, float]  
A type safe version of *tuple(rectangle)*.

### 1.7.10 Geometry functions

**gaphas.geometry.distance\_point\_point** (*point1: Tuple[float, float], point2: Tuple[float, float] = (0.0, 0.0)*) → float

Return the distance from point *point1* to *point2*.

```
>>> f"distance_point_point((0,0), (1,1)):.3f"
'1.414'
```

**gaphas.geometry.distance\_point\_point\_fast** (*point1: Tuple[float, float], point2: Tuple[float, float] = (0.0, 0.0)*) → float

Return the distance from point *point1* to *point2*. This version is faster than *distance\_point\_point()*, but less precise.

```
>>> distance_point_point_fast((0,0), (1,1))
2
```

**gaphas.geometry.distance\_rectangle\_point** (*rect: Tuple[float, float, float, float], point: Tuple[float, float]*) → float

Return the distance (fast) from a rectangle (*x, y, width, height*) to a point.

```
>>> distance_rectangle_point(Rectangle(0, 0, 10, 10), (11, -1))
2
>>> distance_rectangle_point((0, 0, 10, 10), (11, -1))
2
>>> distance_rectangle_point((0, 0, 10, 10), (-1, 11))
2
```

**gaphas.geometry.point\_on\_rectangle** (*rect: Tuple[float, float, float, float], point: Tuple[float, float], border: bool = False*) → Tuple[float, float]

Return the point on which *point* can be projecten on the rectangle. *border = True* will make sure the point is bound to the border of the reactangle. Otherwise, if the point is in the rectangle, it's okay.

```
>>> point_on_rectangle(Rectangle(0, 0, 10, 10), (11, -1))
(10, 0)
>>> point_on_rectangle((0, 0, 10, 10), (5, 12))
(5, 10)
>>> point_on_rectangle(Rectangle(0, 0, 10, 10), (12, 5))
(10, 5)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (3, 4))
(3, 4)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (0, 3))
(1, 3)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (4, 3))
(4, 3)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (4, 9), border=True)
(4, 11)
>>> point_on_rectangle((1, 1, 10, 10), (4, 6), border=True)
```

(continues on next page)

(continued from previous page)

```

(1, 6)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (5, 3), border=True)
(5, 1)
>>> point_on_rectangle(Rectangle(1, 1, 10, 10), (8, 4), border=True)
(11, 4)
>>> point_on_rectangle((1, 1, 10, 100), (5, 8), border=True)
(1, 8)
>>> point_on_rectangle((1, 1, 10, 100), (5, 98), border=True)
(5, 101)

```

`gaphas.geometry.distance_line_point` (*line\_start*: *Tuple*[float, float], *line\_end*: *Tuple*[float, float], *point*: *Tuple*[float, float]) → *Tuple*[float, *Tuple*[float, float]]

Calculate the distance of a point from a line. The line is marked by begin and end point *line\_start* and *line\_end*.

A tuple is returned containing the distance and point on the line.

```

>>> distance_line_point((0., 0.), (2., 4.), point=(3., 4.))
(1.0, (2.0, 4.0))
>>> distance_line_point((0., 0.), (2., 4.), point=(-1., 0.))
(1.0, (0.0, 0.0))
>>> distance_line_point((0., 0.), (2., 4.), point=(1., 2.))
(0.0, (1.0, 2.0))
>>> d, p = distance_line_point((0., 0.), (2., 4.), point=(2., 2.))
>>> f"{d:.3f}"
'0.894'
>>> f"({p[0]:.3f}, {p[1]:.3f})"
'(1.200, 2.400)'

```

`gaphas.geometry.intersect_line_line` (*line1\_start*: *Tuple*[float, float], *line1\_end*: *Tuple*[float, float], *line2\_start*: *Tuple*[float, float], *line2\_end*: *Tuple*[float, float]) → *Optional*[*Tuple*[float, float]]

Find the point where the lines (segments) defined by (*line1\_start*, *line1\_end*) and (*line2\_start*, *line2\_end*) intersect. If no intersection occurs, *None* is returned.

```

>>> intersect_line_line((3, 0), (8, 10), (0, 0), (10, 10))
(6, 6)
>>> intersect_line_line((0, 0), (10, 10), (3, 0), (8, 10))
(6, 6)
>>> intersect_line_line((0, 0), (10, 10), (8, 10), (3, 0))
(6, 6)
>>> intersect_line_line((8, 10), (3, 0), (0, 0), (10, 10))
(6, 6)
>>> intersect_line_line((0, 0), (0, 10), (3, 0), (8, 10))
>>> intersect_line_line((0, 0), (0, 10), (3, 0), (3, 10))

```

Ticket #168: >>> intersect\_line\_line((478.0, 117.0), (478.0, 166.0), (527.5, 141.5), (336.5, 139.5)) (478.5, 141.48167539267016) >>> intersect\_line\_line((527.5, 141.5), (336.5, 139.5), (478.0, 117.0), (478.0, 166.0)) (478.5, 141.48167539267016)

This is a Python translation of the `lines_intersect`, C Code from Graphics Gems II, Academic Press, Inc. The original routine was written by Mukesh Prasad.

EULA: The Graphics Gems code is copyright-protected. In other words, you cannot claim the text of the code as your own and resell it. Using the code is permitted in any program, product, or library, non-commercial or commercial. Giving credit is not required, though is a nice gesture. The code comes as-is, and if there

are any flaws or problems with any Gems code, nobody involved with Gems - authors, editors, publishers, or webmasters - are to be held responsible. Basically, don't be a jerk, and remember that anything free comes with no guarantee.

`gaphas.geometry.rectangle_contains` (*inner: Tuple[float, float, float, float], outer: Tuple[float, float, float, float]*) → bool

Returns True if inner rect is contained in outer rect.

`gaphas.geometry.rectangle_intersects` (*recta: Tuple[float, float, float, float], rectb: Tuple[float, float, float, float]*) → bool

Return True if recta and rectb intersect.

```
>>> rectangle_intersects((5,5,20, 20), (10, 10, 1, 1))
True
>>> rectangle_intersects((40, 30, 10, 1), (1, 1, 1, 1))
False
```

`gaphas.geometry.rectangle_clip` (*recta: Tuple[float, float, float, float], rectb: Tuple[float, float, float, float]*) → Optional[Tuple[float, float, float, float]]

Return the clipped rectangle of recta and rectb. If they do not intersect, None is returned.

```
>>> rectangle_clip((0, 0, 20, 20), (10, 10, 20, 20))
(10, 10, 10, 10)
```

Finally there are classes and modules that make up the building blocks on which Gaphas is built:

## 1.8 Quadtree

In order to find items and handles fast on a 2D surface, a geometric structure is required.

There are two popular variants: [Quadtrees](#) and [R-trees](#). R-trees are tough and well suited for non-moving data. Quadtrees are easier to understand and easier to maintain.

Idea:

- Divide the view in 4 quadrants and place each item in a quadrant.
- When a quadrant has more than 'x' elements, divide it again.
- When an item overlaps more than one quadrant, it's added to the owner.

Gaphas uses item bounding boxed to determine where items should be put.

It is also possible to relocate or remove items to the tree.

The Quadtree itself is added as part of Gaphas' View. The view is aware of item's bounding boxes as it is responsible for user interaction. The Quadtree is set to the size of the window. As a result items which are part of the diagram, may be placed outside the window and thus will not be added to the quadtree. Item's that are partly in- and partly outside the window will be clipped.



### 1.8.1 Interface

The Quadtree interface is simple and tailored towards the use cases of gaphas.

Important properties:

- `bounds`: boundaries of the canvas/view

Methods for working with items in the quadtree:

- `add(item, bounds)`: add an item to the quadtree
- `remove(item)`: remove item from the tree
- `update(item, new_bounds)`: replace an item in the quadtree, using it's new boundaries.
- Multiple ways of finding items have been implemented: 1. Find item closest to point 2. Find all items within distance  $d$  of a point 3. Find all items inside a rectangle 4. Find all items inside or intersecting with a rectangle

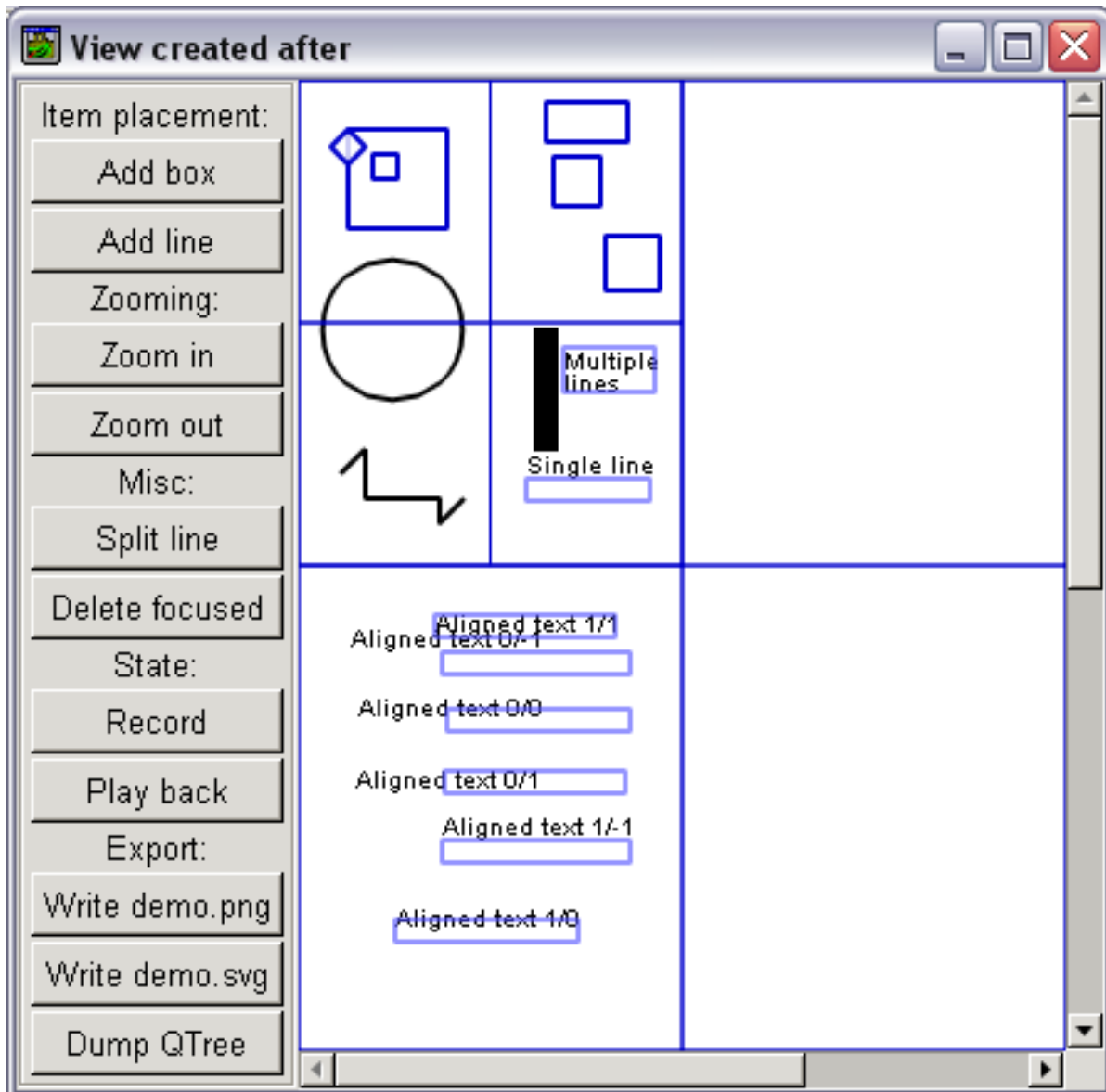
Methods working on the quadtree itself:

- `resize(new_bounds)`: stretch the boundaries of the quadtree if necessary.

### 1.8.2 Implementation

The implementation of gaphas' Quadtree can be found at <https://github.com/gaphor/gaphas/blob/master/gaphas/quadtree.py>.

Here's an example of the Quadtree in action (Gaphas' demo app with `gaphas.view.DEBUG_DRAW_QUADTREE` enabled):



The screen is divided into four equal quadrants. The first quadrant has many items, therefore it has been divided again.

## References

(!PyGame) <http://www.pygame.org/wiki/QuadTree?parent=CookBook>

(PythonCAD) <https://sourceforge.net/p/pythoncad/code/ci/master/tree/PythonCAD/Generic/quadtree.py>

## 1.9 Table

Table is an internal structure. It can be best compared to a table in a database. On the table, indexes can be defined.

Tables are used when data should be made available in different forms.

Source code: <https://github.com/gaphor/gaphas/blob/master/gaphas/table.py>.

## 1.10 Tree

Tree is an internal structure used by the default view model implementation (`gaphas.Canvas`). A tree consists of nodes.

The tree is optimized for depth-first search.

Source code: <https://github.com/gaphor/gaphas/blob/master/gaphas/tree.py>.

## 1.11 Decorators

**class** `gaphas.decorators.g_async` (*single: bool = False, timeout: int = 0, priority: int = `gi.repository.GLib.PRIORITY_DEFAULT`*)

Instead of calling the function, schedule an idle handler at a given priority. This requires the async'ed method to be called from within the GTK main loop. Otherwise the method is executed directly.

---

**Note:** the current implementation of async single mode only works for methods, not functions.

---

Calling the async function from outside the gtk main loop will yield immediate execution:

async just works on functions (as long as `single=False`):

```
>>> a = g_async() (lambda: 'Hi')
>>> a()
'Hi'
```

Simple method:

```
>>> class A(object):
...     @g_async(single=False, priority=GLib.PRIORITY_HIGH)
...     def a(self):
...         print('idle-a', GLib.main_depth())
```

Methods can also set single mode to True (the method is only scheduled once).

```
>>> class B(object):
...     @g_async(single=True)
...     def b(self):
...         print('idle-b', GLib.main_depth())
```

Also a timeout property can be provided:

```
>>> class C(object):
...     @g_async(timeout=50)
...     def c1(self):
```

(continues on next page)

(continued from previous page)

```
...     print('idle-c1', GLib.main_depth())
...     @g_async(single=True, timeout=60)
...     def c2(self):
...         print('idle-c2', GLib.main_depth())
```

This is a helper function used to test classes A and B from within the GTK+ main loop:

```
>>> def delayed():
...     print("before")
...     a = A()
...     b = B()
...     c = C()
...     c.c1()
...     c.c1()
...     c.c2()
...     c.c2()
...     a.a()
...     b.b()
...     a.a()
...     b.b()
...     a.a()
...     b.b()
...     print("after")
...     GLib.timeout_add(100, Gtk.main_quit)
>>> GLib.timeout_add(1, delayed) > 0 # timeout id may vary
True
>>> from gi.repository import Gtk
>>> Gtk.main()
before
after
idle-a 1
idle-a 1
idle-a 1
idle-b 1
idle-c1 1
idle-c1 1
idle-c2 1
```

As you can see, although `b.b()` has been called three times, it's only executed once.

`gaphas.decorators.nonrecursive` (*func*)

Enforce a function or method is not executed recursively:

```
>>> class A(object):
...     @nonrecursive
...     def a(self, x=1):
...         print(x)
...         self.a(x+1)
>>> A().a()
1
>>> A().a()
1
```

## A

add() (*gaphas.canvas.Canvas* method), 14  
 add\_constraint() (*gaphas.connections.Connections* method), 22  
 add\_controller() (*gaphas.view.GtkView* method), 12  
 add\_handler() (*gaphas.connections.Connections* method), 22  
 add\_handler() (*gaphas.solver.Variable* method), 24  
 all\_dirty\_matrix\_items() (*gaphas.view.GtkView* method), 13  
 append() (*gaphas.painter.PainterChain* method), 19

## B

bounding\_box() (*gaphas.view.GtkView* property), 12  
 bounding\_boxPainter() (*gaphas.view.GtkView* property), 12  
 BoundingBoxPainter (class in *gaphas.painter*), 19

## C

Canvas (class in *gaphas.canvas*), 14  
 connect\_item() (*gaphas.connections.Connections* method), 22  
 connectable() (*gaphas.connector.Handle* property), 21  
 Connections (class in *gaphas.connections*), 22  
 connections() (*gaphas.view.model.Model* property), 13  
 constraint() (*gaphas.connector.LinePort* method), 21  
 constraint() (*gaphas.connector.PointPort* method), 21  
 constraint() (*gaphas.connector.Port* method), 21

## D

dirty() (*gaphas.solver.Variable* method), 24  
 disconnect\_item() (*gaphas.connections.Connections* method), 22  
 distance\_line\_point() (in module *gaphas.geometry*), 27

distance\_point\_point() (in module *gaphas.geometry*), 26  
 distance\_point\_point\_fast() (in module *gaphas.geometry*), 26  
 distance\_rectangle\_point() (in module *gaphas.geometry*), 26  
 draw() (*gaphas.item.Item* method), 14  
 draw() (*gaphas.item.Line* method), 18  
 draw\_head() (*gaphas.item.Line* method), 18  
 draw\_tail() (*gaphas.item.Line* method), 18

## E

Element (class in *gaphas.item*), 17  
 expand() (*gaphas.geometry.Rectangle* method), 25

## F

FreeHandPainter (class in *gaphas.painter*), 19

## G

g\_async (class in *gaphas.decorators*), 31  
 get\_all\_items() (*gaphas.canvas.Canvas* method), 15  
 get\_all\_items() (*gaphas.view.model.Model* method), 13  
 get\_children() (*gaphas.canvas.Canvas* method), 15  
 get\_children() (*gaphas.view.model.Model* method), 13  
 get\_connection() (*gaphas.connections.Connections* method), 23  
 get\_connections() (*gaphas.connections.Connections* method), 23  
 get\_item\_bounding\_box() (*gaphas.view.GtkView* method), 12  
 get\_items\_in\_rectangle() (*gaphas.view.GtkView* method), 12  
 get\_matrix\_i2c() (*gaphas.canvas.Canvas* method), 16  
 get\_matrix\_i2v() (*gaphas.view.GtkView* method), 12  
 get\_matrix\_v2i() (*gaphas.view.GtkView* method), 12

`get_parent()` (*gaphas.canvas.Canvas* method), 15  
`get_parent()` (*gaphas.view.model.Model* method), 13  
`get_root_items()` (*gaphas.canvas.Canvas* method), 15  
`glue()` (*gaphas.connector.LinePort* method), 21  
`glue()` (*gaphas.connector.PointPort* method), 21  
`glue()` (*gaphas.connector.Port* method), 21  
`GtkView` (class in *gaphas.view*), 11

## H

`hadjustment()` (*gaphas.view(GtkView)* property), 12  
`Handle` (class in *gaphas.connector*), 20  
`HandlePainter` (class in *gaphas.painter*), 19  
`handles()` (*gaphas.item.Element* method), 17  
`handles()` (*gaphas.item.Item* method), 14  
`handles()` (*gaphas.item.Line* method), 18  
`height()` (*gaphas.item.Element* property), 17  
`hover_tool()` (in module *gaphas.tool*), 20

## I

`intersect_line_line()` (in module *gaphas.geometry*), 27  
`Item` (class in *gaphas.item*), 14  
`item_tool()` (in module *gaphas.tool*), 20  
`ItemPainter` (class in *gaphas.painter*), 19  
`ItemPainterType` (class in *gaphas.painter.painter*), 18

## L

`Line` (class in *gaphas.item*), 17  
`LinePort` (class in *gaphas.connector*), 21

## M

`Matrix` (class in *gaphas.matrix*), 25  
`matrix()` (*gaphas.item.Item* property), 14  
`matrix()` (*gaphas.view(GtkView)* property), 11  
`matrix_i2c()` (*gaphas.item.Item* property), 14  
`MatrixProjection` (class in *gaphas.position*), 25  
`Model` (class in *gaphas.view.model*), 13  
`model()` (*gaphas.view(GtkView)* property), 12  
`movable()` (*gaphas.connector.Handle* property), 21

## N

`nonrecursive()` (in module *gaphas.decorators*), 32  
`notify()` (*gaphas.solver.Variable* method), 24

## O

`opposite()` (*gaphas.item.Line* method), 18

## P

`paint()` (*gaphas.painter.Painter* method), 18

`paint()` (*gaphas.painter.painter.ItemPainterType* method), 19  
`paint_item()` (*gaphas.painter.painter.ItemPainterType* method), 18  
`Painter` (class in *gaphas.painter*), 18  
`painter()` (*gaphas.view(GtkView)* property), 12  
`PainterChain` (class in *gaphas.painter*), 19  
`placement_tool()` (in module *gaphas.tool*), 20  
`point()` (*gaphas.item.Element* method), 17  
`point()` (*gaphas.item.Item* method), 14  
`point()` (*gaphas.item.Line* method), 18  
`point_on_rectangle()` (in module *gaphas.geometry*), 26  
`PointPort` (class in *gaphas.connector*), 21  
`Port` (class in *gaphas.connector*), 21  
`ports()` (*gaphas.item.Element* method), 17  
`ports()` (*gaphas.item.Item* method), 14  
`ports()` (*gaphas.item.Line* method), 18  
`pos()` (*gaphas.connector.Handle* property), 20  
`Position` (class in *gaphas.position*), 24  
`prepend()` (*gaphas.painter.PainterChain* method), 19

## Q

`queue_redraw()` (*gaphas.view(GtkView)* method), 12

## R

`reconnect_item()` (*gaphas.connections.Connections* method), 23  
`Rectangle` (class in *gaphas.geometry*), 25  
`rectangle_clip()` (in module *gaphas.geometry*), 28  
`rectangle_contains()` (in module *gaphas.geometry*), 28  
`rectangle_intersects()` (in module *gaphas.geometry*), 28  
`register_view()` (*gaphas.canvas.Canvas* method), 17  
`register_view()` (*gaphas.view.model.Model* method), 13  
`remove()` (*gaphas.canvas.Canvas* method), 15  
`remove_all_controllers()` (*gaphas.view(GtkView)* method), 12  
`remove_connections_to_item()` (*gaphas.connections.Connections* method), 22  
`remove_constraint()` (*gaphas.connections.Connections* method), 22  
`remove_controller()` (*gaphas.view(GtkView)* method), 12  
`remove_handler()` (*gaphas.connections.Connections* method), 22  
`remove_handler()` (*gaphas.solver.Variable* method), 24  
`reparent()` (*gaphas.canvas.Canvas* method), 15

[request\\_matrix\\_update\(\)](#) ([gaphas.canvas.Canvas method](#)), 16  
[request\\_update\(\)](#) ([gaphas.canvas.Canvas method](#)), 16  
[request\\_update\(\)](#) ([gaphas.view.GtkView method](#)), 12  
[request\\_update\(\)](#) ([gaphas.view.model.Model method](#)), 13  
[rubberband\\_tool\(\)](#) ([in module gaphas.tool](#)), 20  
[RubberbandPainter](#) ([class in gaphas.tool.rubberband](#)), 19

## S

[scroll\\_tool\(\)](#) ([in module gaphas.tool](#)), 20  
[selection\(\)](#) ([gaphas.view.GtkView property](#)), 12  
[solve\(\)](#) ([gaphas.connections.Connections method](#)), 22  
[solver\(\)](#) ([gaphas.connections.Connections property](#)), 22  
[sort\(\)](#) ([gaphas.canvas.Canvas method](#)), 16  
[sort\(\)](#) ([gaphas.view.model.Model method](#)), 13  
[strength\(\)](#) ([gaphas.solver.Variable property](#)), 24

## T

[tuple\(\)](#) ([gaphas.geometry.Rectangle method](#)), 26

## U

[unregister\\_view\(\)](#) ([gaphas.canvas.Canvas method](#)), 17  
[unregister\\_view\(\)](#) ([gaphas.view.model.Model method](#)), 14  
[update\\_bounding\\_box\(\)](#) ([gaphas.view.GtkView method](#)), 13  
[update\\_now\(\)](#) ([gaphas.canvas.Canvas method](#)), 16  
[update\\_now\(\)](#) ([gaphas.view.model.Model method](#)), 13  
[update\\_orthogonal\\_constraints\(\)](#) ([gaphas.item.Line method](#)), 17

## V

[vadjustment\(\)](#) ([gaphas.view.GtkView property](#)), 12  
[Variable](#) ([class in gaphas.solver](#)), 24  
[variable\(\)](#) ([in module gaphas.solver](#)), 24  
[visible\(\)](#) ([gaphas.connector.Handle property](#)), 21

## W

[width\(\)](#) ([gaphas.item.Element property](#)), 17

## Z

[zoom\(\)](#) ([gaphas.view.GtkView method](#)), 12  
[zoom\\_tool\(\)](#) ([in module gaphas.tool](#)), 20