Gaphas Documentation

Arjan J. Molenaar

Apr 22, 2024

THE BASICS

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

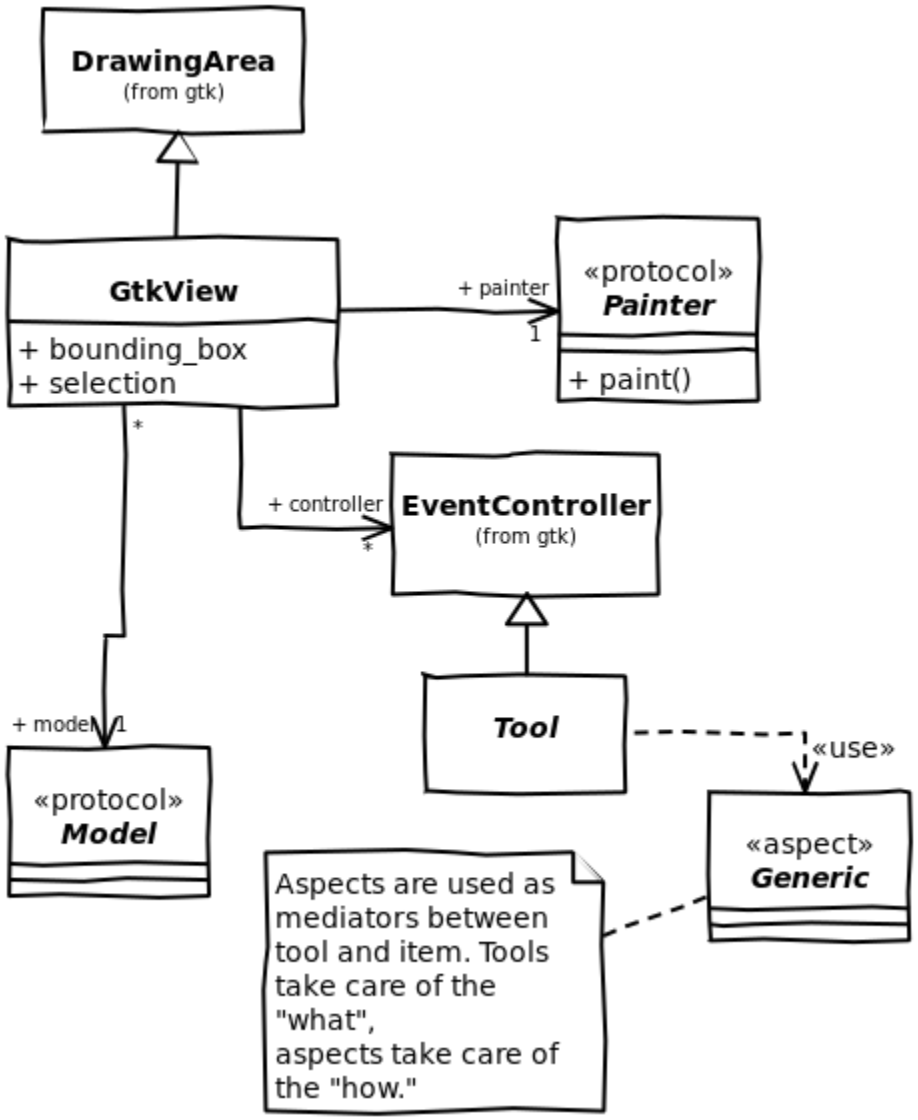
CHAPTER

ONE

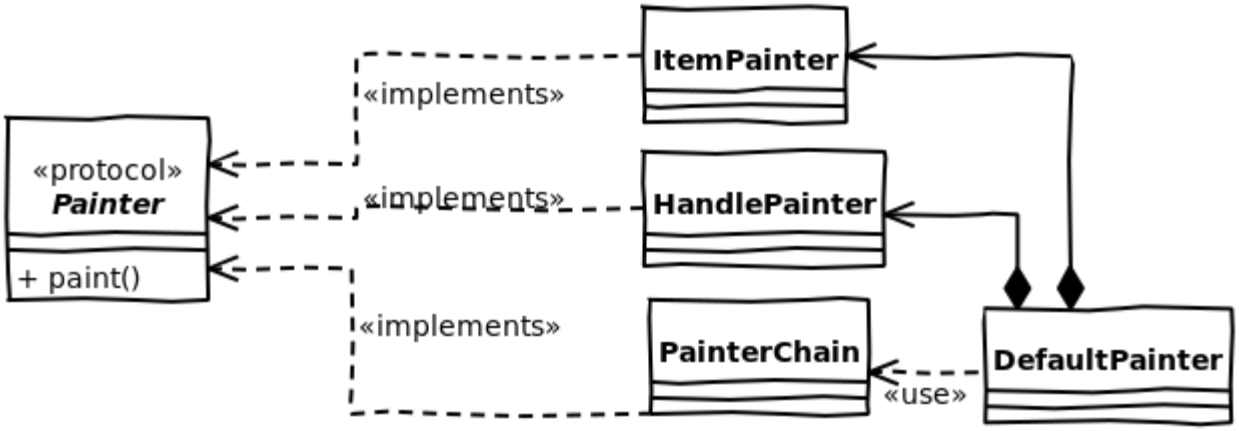
CLASS DIAGRAM

This class diagram describes the basic layout of Gaphas.

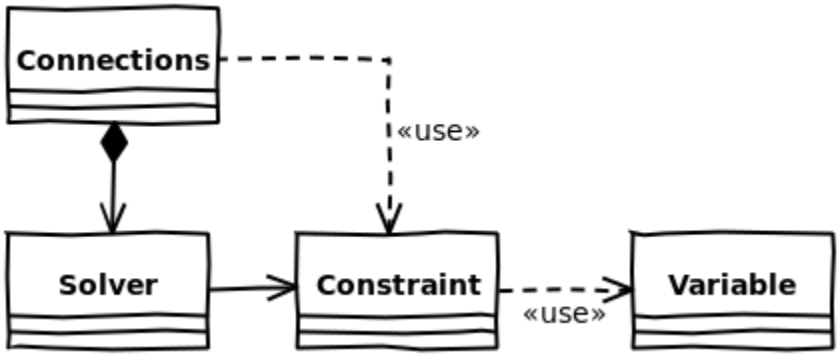
The central class is `GtkView`. It takes a model. A default implementation is provided by *gaphas.Canvas*. A view is rendered by Painters. Interaction is handled by Tools.



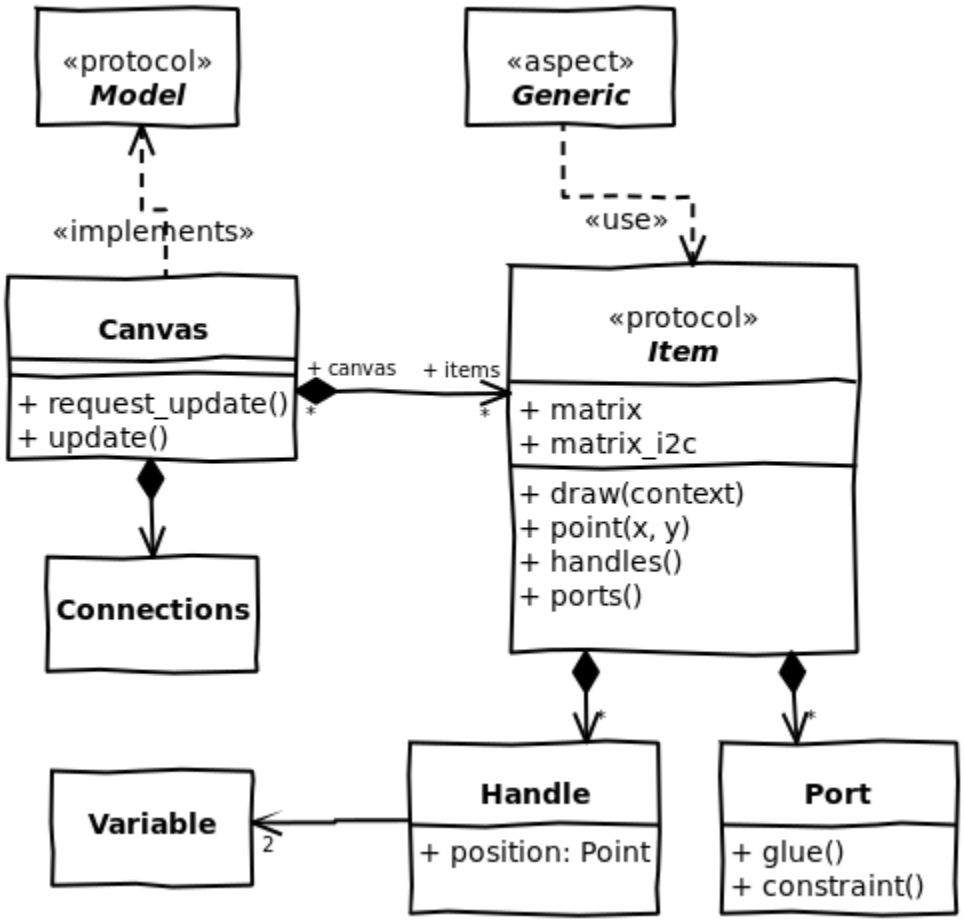
Painting is done by painters. Each painter will paint a layer of the canvas.



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.


«protocol»
Model
«aspect»
Generic
«implements»
«use»
Canvas
+ request_update()
+ update()
+ canvas
+ items
*
*
«protocol»
Item
+ matrix
+ matrix_i2c
+ draw(context)
+ point(x, y)
+ handles()
+ ports()
Connections
Variable
2
Handle
+ position: Point
Port
+ glue()
+ constraint()

CHAPTER

TWO

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[1].

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.:

- 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 hit 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`

[1] as opposed to versions < 0.5, where tools could be shared among multiple views.

item tool
Items are the elements that are actually providing any (visual) meaning to the diagram. ItemTool 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[2].

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.

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

CHAPTER

THREE

CONNECTIONS

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

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 rectanglular 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)))
```

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

CHAPTER

FOUR

CONSTRAINT SOLVER

Gaphas' constraint solver can be consider 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.

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.

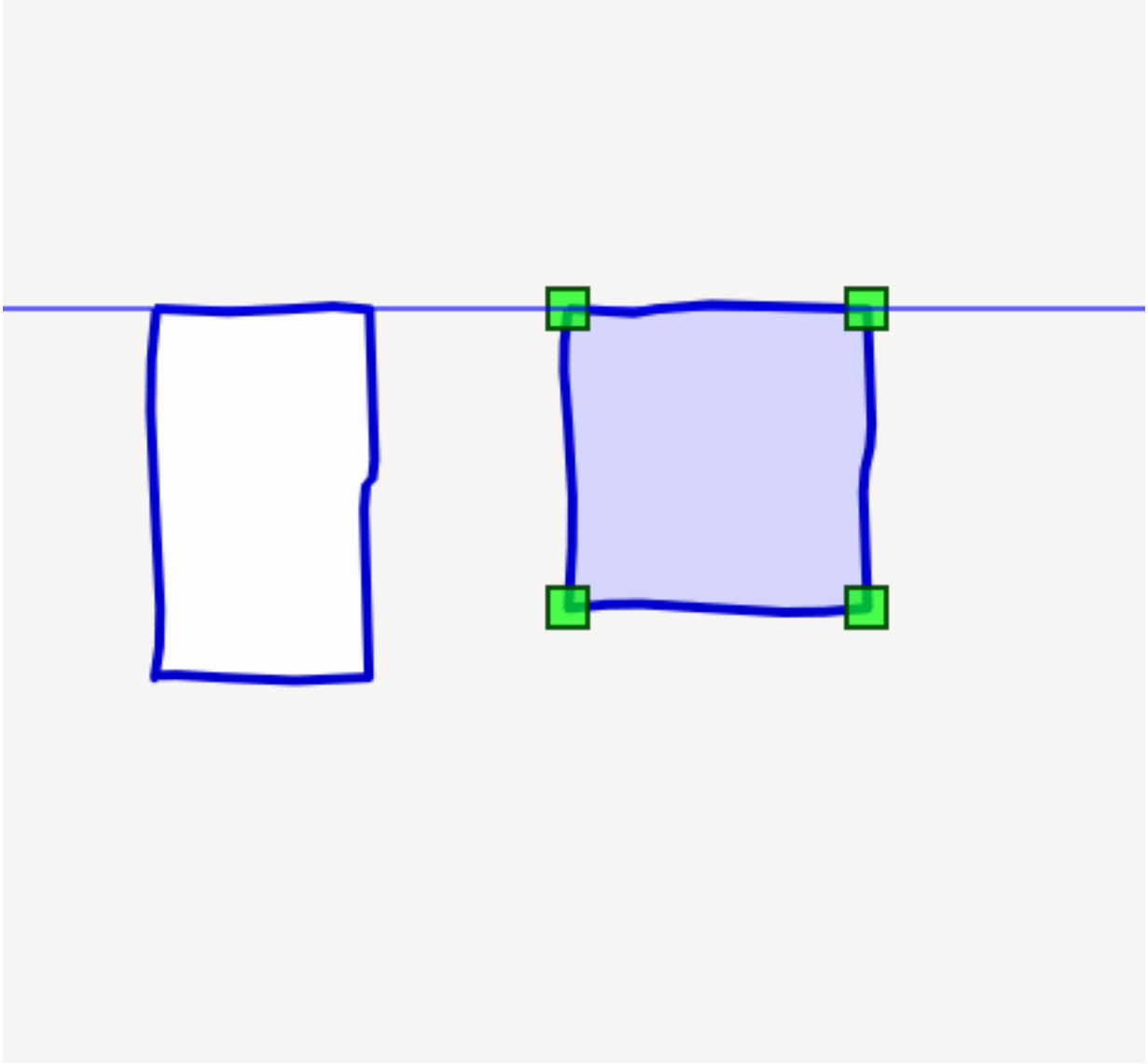
The Solver can be found at: https://github.com/gaphor/gaphas/blob/main/gaphas/solver/, along with Variable and the Constraint base class.

CHAPTER

FIVE

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, 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.add_controller(zoom_tool())
```

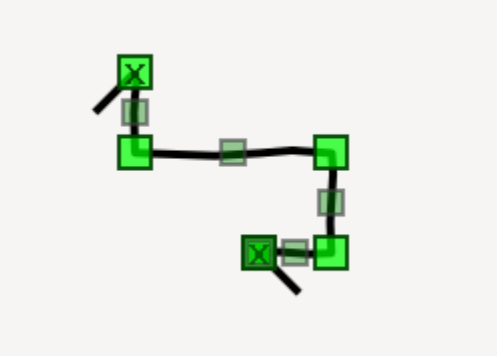
You need to hook up the `GuidePainter`. The aspect are loaded as soon as the module is loaded.

CHAPTER

SIX

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. Splitting and merging of lines is supported by `item_tool`, however to actually use it, the `segment` module needs to be imported.

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

CHAPTER

SEVEN

API REFERENCE

7.1 View

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

7.2 Model

7.2.1 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.model.**Model**(**args*, ***kwargs*)

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

property connections: *Connections*

The connections instance used for this model.

get_all_items() → Iterable[*Item*]

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

Normally that will be depth-first.

get_parent(*item:* Item) → *Item* | None

Get the parent item of an item.

Returns `None` if there is no parent item.

get_children(*item:* Item | *None*) → Iterable[*Item*]

Iterate all direct child items of an item.

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

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

request_update(*item:* Item) → None

Request update for an item.

Parameters

item (`Item`) – The item to be updated

update_now(*dirty_items: Collection[*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: View*) → None

Allow a view to be registered.

Registered views should receive update requests for modified items.

unregister_view(*view: 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: *Matrix*

The "local", item-to-parent matrix.

property matrix_i2c: *Matrix*

Matrix from item to toplevel.

handles() → Sequence[*Handle*]

Return a list of handles owned by the item.

ports() → Sequence[*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: 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)

7.2.2 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
>>> 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) → *Item* | None

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:* Item | *None*) → 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*

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) → 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: float

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

property height: float

Height.

handles() → Sequence[*Handle*]

Return a list of handles owned by the item.

ports() → Sequence[*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() → None

Update the constraints required to maintain the orthogonal line.

opposite(*handle:* Handle) → *Handle*

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

handles() → Sequence[*Handle*]

Return a list of handles owned by the item.

ports() → Sequence[*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: DrawContext*) → None

Default head drawer: move cursor to the first handle.

draw_tail(*context: DrawContext*) → None

Default tail drawer: draw line to the last handle.

draw(*context: DrawContext*) → None

Draw the line itself.

See Item.draw(context).

7.3 Painters

Painters are used to draw the view.

7.3.1 Protocols

Each painter adheres to the `Painter` protocol.

class gaphas.painter.**Painter**(**args*, ***kwargs*)

Painter interface.

paint(*items: Collection[*Item*]*, *cairo: cairo.Context*) → None

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*, ***kwargs*)

paint_item(*item:* Item, *cairo: cairo.Context*) → None

Draw a single item.

paint(*items: Collection[*Item*]*, *cairo: cairo.Context*) → None

Do the paint action (called from the View).

7.3.2 Default implementations

class gaphas.painter.**PainterChain**

Chain up a set of painters.

append(*painter:* Painter) → *PainterChain*

Add a painter to the list of painters.

prepend(*painter:* Painter) → *PainterChain*

Add a painter to the beginning of the list of painters.

class gaphas.painter.**ItemPainter**(*selection: Selection | None = None*)

class gaphas.painter.**HandlePainter**(*view: GtkView*)

Draw handles of items that are marked as selected in the view.

class gaphas.painter.**FreeHandPainter**(*item_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

7.3.3 Rubberband tool

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

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.

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

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.

7.5.1 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: bool

Can this handle actually connectect to a port?

property movable: bool

Can this handle be moved by a mouse pointer?

property visible: bool

Is this handle visible to the user?

property glued: bool

Is the handle being moved and about to be connected?

7.5.2 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:* Position, *end:* 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:* 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.

7.6 Connections

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

class gaphas.connections.**Connections**(*solver: Solver | None = 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: Solver

The solver used by this connections instance.

solve() → None

Solve all constraints.

add_constraint(*item: Item*, *constraint: Constraint*) → Constraint

Add a "simple" constraint for an item.

remove_constraint(*item: Item*, *constraint: Constraint*) → None

Remove an item specific constraint.

connect_item(*item: Item*, *handle: Handle*, *connected: Item*, *port: Port | None*, *constraint: Constraint | None = None*, *callback: Callable[[Item, Handle, Item, Port], None] | 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: Item*, *handle: Handle | None = 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: 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:* Item, *handle:* Handle, *port:* Port | *None* = *None*, *constraint:* Constraint | *None* = *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:* Handle) → Connection | None

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:* Item | *None* = *None*, *handle:* Handle | *None* = *None*, *connected:* Item | *None* = *None*, *port:* Port | *None* = *None*) → Iterator[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...]
```

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[[*Variable*, float], None]*) → None

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

remove_handler(*handler: Callable[[*Variable*, float], None]*) → None

Remove a handler.

notify(*old: float*) → None

Notify all handlers.

property strength: int

Strength.

dirty() → 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:* Position, *matrix:* Matrix)

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

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: cairo.Matrix | None = None*)

Matrix wrapper.

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

7.9 Rectangle

class gaphas.geometry.**Rectangle**(*x: float = 0*, *y: float = 0*, *width: float | None = None*, *height: float | None = 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)*.

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]* | Rectangle, *point: Tuple[float, float]*) → float

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

gaphas.geometry.**point_on_rectangle**(*rect: Tuple[float, float, float, float]* | Rectangle, *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.

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.

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]*) → Tuple[float, float] | None

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]*) → Tuple[float, float, float, float] | None

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:

CHAPTER

EIGHT

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 size is defined by its contents.

8.1 Interface

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

Important properties:

- bounds: boundaries of the canvas

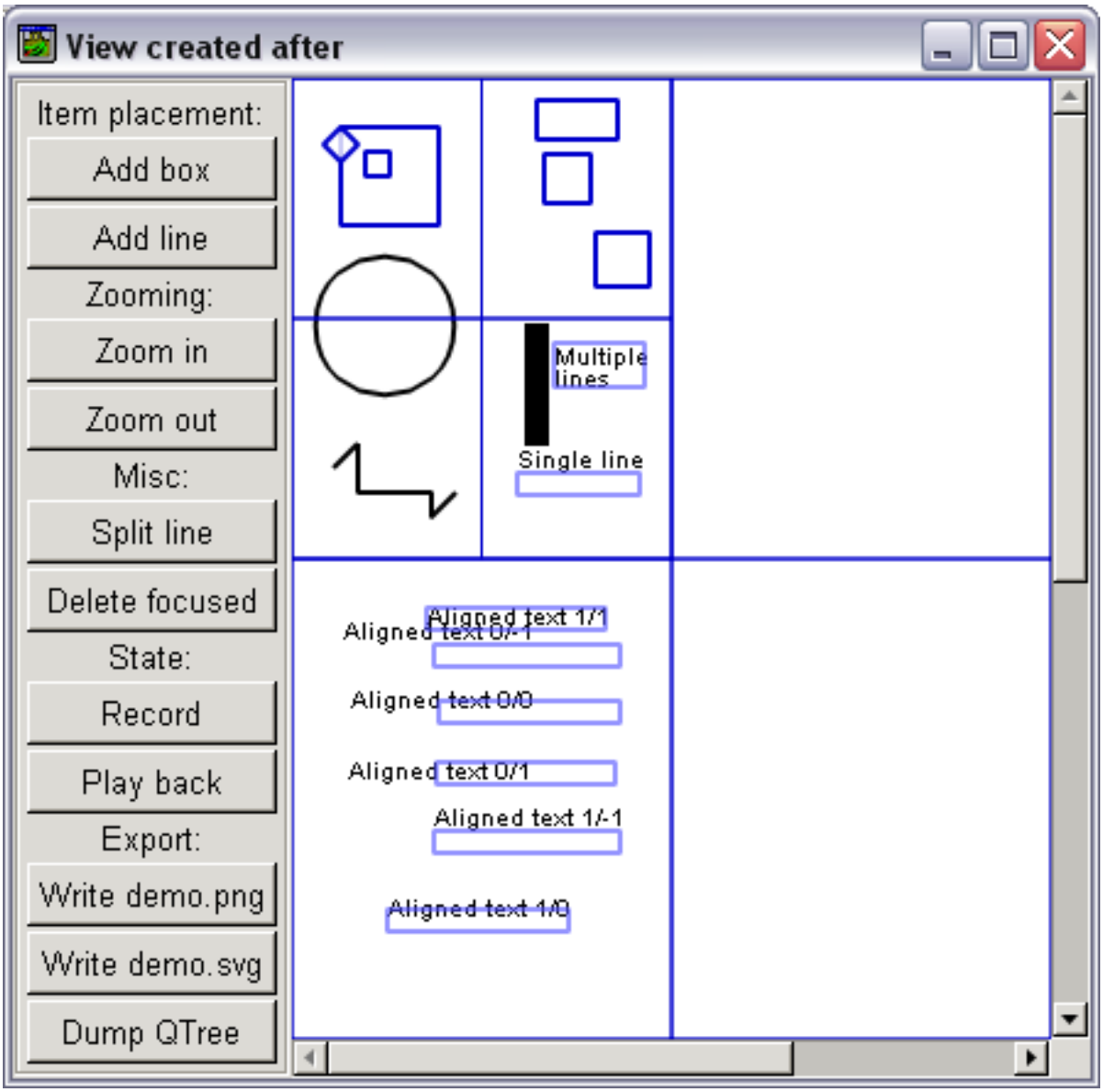
Methods for working with items in the quadtree:

- *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

8.2 Implementation

The implementation of gaphas' Quadtree can be found at https://github.com/gaphor/gaphas/blob/main/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.

8.2.1 References

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

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

CHAPTER

NINE

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/main/gaphas/table.py.

CHAPTER

TEN

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/main/gaphas/tree.py.

CHAPTER

ELEVEN

DECORATORS

class gaphas.decorators.**g_async**(*single: bool = False*, *timeout: int = 0*, *priority: int = 200*)

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.

If a function's first argument is "self", it's considered a method.

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

A function can also be a generator. The generator will be fully executed. If run in the main loop, an empty iterator will be returned. A generator is "single" by default. Because of the nature of generators the first invocation will run till completion.

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
```

INDEX

M

N

O

P

R

S

T

U

V

W