bear$_{h}ug$

Release 0.2

Nov 01, 2020

Contents:

CHAPTER 1

bear_hug

1.1 bear_hug package

1.1.1 Submodules

bear_hug.bear_hug module

An object-oriented bearlibterminal wrapper with the support for complex ASCII art and widget-like behaviour.

class bear_hug.bear_hug.**WidgetLocation**(*pos*, *layer*)

Bases: tuple

layer

Alias for field number 1

pos

Alias for field number 0

class bear_hug.bear_hug.**BearTerminal**(*font_path='../demo_assets/cp437_12x12.png'*, ***kwargs*)

Bases: object

A main terminal class.

This class corresponds to a single window and is responsible for drawing whatever widgets get added to this window, as well as processing any input.

Accepts bearlibterminal library configuration options as kwargs to `self.__init__`. Currently only library settings are supported and there is no support for changing them on the fly.

start()

Open a terminal and place it on the screen.

Library settings that were passed as kwargs to *self.__init__()* get actually applied during when this method is executed.

clear()
Remove all widgets from this terminal, but do not close it.

refresh()
Refresh a terminal.

Actually draws whatever changes were made by `*_widget` methods.

close()
Close a terminal.

Does not destroy Widget objects or call any other cleanup routine.

add_widget(*widget*, *pos=(0, 0)*, *layer=0*, *refresh=False*)
Add a widget to the terminal and set *widget.terminal* to *self*.

No two widgets are allowed to overlap within a layer and no widget can be added twice.

Parameters

- **widget** – a Widget instance
- **pos** – top left corner of the widget
- **layer** – layer to place the widget on
- **refresh** – whether to refresh terminal after adding the widget. If False, the widget will not be actually shown until the next `terminal.refresh()` call

remove_widget(*widget*, *refresh=False*)
Remove widget from the terminal.

This method does not cause or imply the destruction of Widget object; it merely removes it from the terminal.

Parameters

- **widget** – A widget to be removed
- **refresh** – whether to refresh the terminal after removing a widget. If False, the widget will be visible until the next `terminal.refresh()` call

move_widget(*widget*, *pos*, *refresh=False*)
Move widget to a new position.

Widgets can only be moved within the layer. If it is necessary to move a widget from one layer to another, it should be removed and added anew.

Parameters

- **widget** – A widget to be moved
- **pos** –

  param refresh whether to refresh the terminal after removing a widget. If False, the widget won't move on screen until the next `terminal.refresh()` call

update_widget(*widget*, *refresh=False*)
Actually draw widget chars on screen.

If `widget.chars` or `widget.colors` have changed, this method will make these changes visible. It is also called by `self.add_widget()` and other methods that have a `refresh` argument.

Parameters **widget** – A widget to be updated.

get_widget_by_pos(*pos*, *layer=None*)
Return the widget currently placed at the given position.

Parameters

- **pos** – Position (a 2-tuple of ints)
- **layer** – A layer to look at. If this is set to valid layer number, returns the widget (if any) from that layer. If not set, return the widget from highest layer where a given cell is non-empty.

check_input()

Check if terminal has input. If so, yield corresponding `BearEvent`.

This method returns an iterator because it's possible there would be more than one event in a single tick, eg when two buttons are pressed simultaneously.

This method mostly just wraps bearlibterminal's input behaviour in **'events<foo.wyrd.name/en:bearlibterminal:reference:input>'_**, with a single exception: in bearlibterminal, when a key is pressed and held for more than a single tick, it first emits key_down, then waits for 0.5 seconds. Then, if the key is not released, it assumes the key is indeed held and starts spamming events every tick. This makes sense to avoid messing up the typing (where a slow typist would get char sequences like tthiisss).

Bearlibterminal, on the other hand, is meant mostly for games that require more precise input timing. Therefore, it starts spamming `key_down` events immediately after the button is pressed and expects widgets and listeners to mind their input cooldowns themselves.

Yields BearEvent instances with `event_type` set to `misc_input`, `key_up` or `key_down`.

check_state(*query*)

Wrap BLT state

Accepts any of the `TK_*` strings and returns whatever `terminal.state` has to say about it.

Parameters **query** – query string

class bear_hug.bear_hug.**BearLoop**(*terminal*, *queue*, *fps=30*, *profile=False*)

Bases: `object`

A loop that passes events around every 1/fps seconds.

Every tick, the loop calls its `run_iteration()` method, adding tick-related and input-related events to the queue, and then forcing it to start passing all events to the correct subscribers.

There are two tick-related events. In the beginning of the tick it's `tick`-type event whose value is time since the last similar event (in seconds). This is guaranteed to be emitted before any other events from this tick, since the queue wouldn't finish the previous one until it was empty.

In the end of the tick it's a `service`-type event with the value 'tick_over', which is emitted after the entire queue has been processed. It is meant to let subscribers know that the tick is over and nothing is going to happen until the next one. This is, for example, a perfect moment for a Layout to redraw itself, or for a logger to write everything down.

If any events are emitted in response to this event, they will be passed around before the next `tick`. This is a great source of bugs, so it is not advised to respond to `tick_over` unless absolutely necessary.

The loop cannot be started until it has a valid terminal. When the loop is stopped, this terminal is shut down.

Parameters

- **terminal** – a BearTerminal instance to collect input from.
- **queue** – a bear_hug.event.BearEventDispatcher instance to send events to.
- **fps** – a number of times per second this loop should process events.

run()
Start a loop.

It would run until stopped with `self.stop()`

stop()
Order the loop to stop.

It would not actually do it until the current tick is processed.

bear_hug.bear_utilities module

A collection of random stuff for bear_hug that wouldn't fit into other submodules.

Includes a series of useful functions and all bear_hug exception classes.

bear_hug.bear_utilities.**shapes_equal**(*a*, *b*)
Tests if two nested lists are of the same shape

Parameters

- **a** – list
- **b** – list

Returns True if lists are indeed of the same shape, False otherwise

bear_hug.bear_utilities.**copy_shape**(*l*, *value=None*)
Takes a nested list and returns the new list of the same shape, completely filled with the same value.

May cause bugs when the value is mutable (for example, a list) because it fills the returned list with (pointers to) the same element, not independent copies. Since in practice this function is used to create `colors` for a widget with known `chars`, or otherwise to mess around with chars/colors data (which are normally replaced entirely, not edited), it is left for the callers to make sure their values are OK.

Parameters

- **l** – initial list
- **value** – value to fill the list with

bear_hug.bear_utilities.**slice_nested**(*l*, *slice_pos*, *slice_size*)
Slice the nested list

Parameters

- **l** – a nested list
- **slice_pos** – a 2-tuple (x, y) of slice start
- **slice_size** – a 2-tuple (width, height) of slice size

bear_hug.bear_utilities.**rotate_list**(*l*)
Take a nested list of (x, y) dimensions, return an (y, x) list.

Parameters **l** – a 2-nested list

bear_hug.bear_utilities.**rectangles_collide**(*pos1*, *size1*, *pos2*, *size2*)
Return True if the rectangles collide

Rectangles are supplied in [x,y], [xsize, ysize] form with the left corner and size. Assumes positions and sizes to be sorted

Parameters

- **pos1** – top left corner of the first rectangle, as (x, y) 2-tuple

- **size1** – size of the first rectangle, as (width, height) 2-tuple
- **pos2** – top left corner of the second rectangle, as (x, y) 2-tuple
- **size2** – size of the second rectangle, as (width, height) 2-tuple

bear_hug.bear_utilities.**has_values**(*l*)
Returns True if a 2-nested list contains at least one truthy value.

Parameters **l** – a nested list

Returns

bear_hug.bear_utilities.**blit**(*l1*, *l2*, *x*, *y*)
Blits `l2` to `l1` at a given pos, overwriting the original values.

This method does not actually affect `l1`; instead, it copies it to a new variable, sets whatever values need to be set, and returns the modified copy.

Parameters

- **l1** – A 2-nested list.
- **l2** – A 2-nested list.
- **y** (*x,*) – A top left corner of `l2` relative to `l1`.

Returns

bear_hug.bear_utilities.**generate_box**(*size*, *line_width='single'*)
Generate a chars list for a box bounded by pseudographic lines.

Uses CP437 chars 0xB3-0xDA translated to Unicode points (see **'here<http://www.unicode.org/Public/MAPPINGS/VENDOR**
for the translation table)

Parameters

- **size** – an (xsize, ysize) tuple
- **line_width** – str. Either 'single' or 'double'

Returns a nested list of chars.

class bear_hug.bear_utilities.**Singleton**
Bases: `type`

A Singleton metaclass for EntityTracker

exception bear_hug.bear_utilities.**BearException**
Bases: `Exception`

A base class for all bear_hug exceptions

exception bear_hug.bear_utilities.**BearLoopException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with the loop or event system.

exception bear_hug.bear_utilities.**BearLayoutException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with adding/drawing/removing a Widget on a Layout

exception bear_hug.bear_utilities.**BearECSException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with Entity-Component System.

exception bear_hug.bear_utilities.**BearSoundException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with the sound.

exception bear_hug.bear_utilities.**BearJSONException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with JSON (de)serialization of widgets or entities.

exception bear_hug.bear_utilities.**BearResourceException**
Bases: *bear_hug.bear_utilities.BearException*

Something wrong with loading ASCII assets

bear_hug.ecs module

Entity-component system.

Entities are just an ID and the container of components. The major way for them to do something useful should be components calling something like `self.owner.that_other_component.do_stuff()` or emitting events.

The creation of a new Entity is announced by the following event: `BearEvent(event_type='ecs_create', event_value=entity)`

It is an only event type that uses the actual entity object, not its ID, as `event_value`. When this event is emitted, the entity should be ready to work; in particular, all its components should be subscribed to the appropriate events.

Both Entities and Components can be serialized to JSON using `repr(object)` and then deserialized.

class bear_hug.ecs.**Entity**(*id='Default ID'*, *components=[]*)
Bases: `object`

A root entity class.

This is basically a container of components, and an ID.

Entity ID not checked for uniqueness during Entity creation, because it's possible that the Entity object will be created before the queue is turned on (and, therefore, before EntityTracker knows anything about any entities), but having non-unique IDs is practically guaranteed to cause some entertaining glitches.

When the component is added to the Entity, its name (a `component.name` attribute) is added to `entity.__dict__`. This way, other components can then address it as `self.owner.position` or `self.owner.widget` or whatever. Names thus serve as something like slots, so that an entity couldn't have multiple components for the same function. Possible names are not restricted in any way, but it is strongly recommended not to change them during inheritance between Component subclasses, and especially not to use the same name for any two components that could ever possibly be used within a single entity.

Parameters

- **id** – a string used as an Entity ID.
- **components** – an iterable of Component instances that can will be added to this entity.

add_component(*component*)
Add a single component to the Entity.

Raises exception if `Component.name` is already in `self.__dict__` and not in `self.components`. This allows overwriting components (should you want to change eg the entity's widget), while protecting the non-Component properties.

Parameters **component** – A Component instance.

remove_component(*component_name*)
Remove a single component from this entity.

Uses the `Component.name`, not an actual instance, as an argument. If the Entity doesn't have such a component, raises `BearECSException`

Parameters **component_name** – The name of a component to remove

Returns

class bear_hug.ecs.**Component**(*dispatcher*, *name='Root'*, *owner=None*)
Bases: *bear_hug.widgets.Listener*

A root component class.

Component name is expected to be the same between all components of the same general type (normally, base class for a given role, like position component, AI/input controller or a widget interface) and all its subclasses). Component inherits from Listener and is therefore able to receive and return `BearEvent`. Of course, it needs the correct subscriptions to actually get them.

`repr(component)` is used for serialization and should generate a valid JSON-encoded dict. It should always include a 'class' key which should equal the class name for that component and will be used by a deserializer to determine what to create. All other keys will be deserialized and treated as kwargs to a newly-created object. To define the deserialization protocol, JSON dict may also contain keys formatted as `{kwarg_name}_dtype` which should be a string and will be eval-ed during deserialization. Only Python's builtin converters (eg `str`, `int` or `float`) are allowed; custom ones are currently unsupported.

For example, the following is a valid JSON:

```
` {"class": "TestComponent", "x": 5, "y": 5, "direction": "r",
"former_owners": ["asd", "zxc", "qwe"], "former_owners_type": "set"}
`
```

Its deserialization is equivalent to the following call:

```
x = TestComponent(x=5, y=5, direction='r', former_owners=set(['asd',
'zxc', 'qwe']))
```

The following keys are forbidden: 'name', 'owner', 'dispatcher'. Kwarg validity is not controlled except by `Component.__init__()`.

Parameters

- **dispatcher** – A queue that the component should subscribe to. `Component.__init__()` may use this to subscribe to whatever events it needs.
- **name** – A name that will be added to `Entity.__dict__`. Should be hardcoded in all Component subclasses.
- **owner** – the Entity (actual object, not ID) to which this object should attach.

set_owner(*owner*)
Register a component owner.

This is only useful if the component is passed from one owner to another, or if the component is created with the *owner* argument (thus attaching it immediately upon creation). This method calls owner's `add_component`

Parameters **owner** – an Entity to attach to.

on_event(*event*)
Component's event callback. Should be overridden if subclasses want to process events.

Parameters **event** – BearEvent instance

class bear_hug.ecs.**EntityTracker**
Bases: *bear_hug.widgets.Listener*

A singleton Listener that keeps track of all existing entities.

Listens to the `ecs_add` and `ecs_destroy events`, updating `self.entities` accordingly.

Can be used to look up an entity by its ID:

```
entity_called_id = EntityTracker.entities['entity_id']
```

Can also be used to get all entities that correspond to some criterion:

```
entity_iter = EntityTracker().filter_entities(lambda x:  'part_of_id' in
x.id)
```

filter_entities(*key=<function EntityTracker.<lambda>>*)
Return all entities for which key evaluates to True.

Note that this method returns entity objects themselves, not the IDs.

Parameters **key** – A single-arg callable

Returns iterator of Entities

class bear_hug.ecs.**WidgetComponent**(*dispatcher*, *widget*, *owner=None*)
Bases: *bear_hug.ecs.Component*

Widget as a component.

This component is an ECS wrapper around the Widget object. Since Widgets can accept events and it is sometimes reasonable to keep some event logic in the Widget instead of Components (ie to keep animation running), its `on_event` method simply passes the events to the Widget. It also supports `height`, `width` and `size` properties, also by calling widget's ones.

Parameters **widget** – A Widget instance.

height
Height of the widget

width
Width of the widget

size
A (width, height) tuple

class bear_hug.ecs.**SwitchWidgetComponent**(**args*, ***kwargs*)
Bases: *bear_hug.ecs.WidgetComponent*

A widget component that supports SwitchingWidget.

Provides methods to use its widget-switching abilities without other components having to call Widget object directly.

switch_to_image(*image_id*)
Switch widget to a necessary image.

If image ID is incorrect, the widget will raise BearException.

Parameters **image_id** – image ID (str)

validate_image(*image_id*)
Return True if image_id is a valid ID for its widget

Parameters **image_id** – image ID (str)

class bear_hug.ecs.**PositionComponent**(*dispatcher*, *x=0*, *y=0*, *vx=0*, *vy=0*, *last_move=(1, 0)*, *affect_z=True*, *owner=None*)

Bases: *bear_hug.ecs.Component*

A component responsible for positioning an Entity on ECSLayout.

Parameters

- **x** – A position of top left corner along X axis.
- **y** – A position of top left corner along Y axis
- **vx** – Horizontal speed (chars per second)
- **vy** – Vertical speed (chars per second)
- **affect_z** – Set Z-level for widgets when placing. Default True

move(*x*, *y*, *emit_event=True*)

Move the Entity to a specified position.

Parameters

- **x** – x
- **y** – y
- **emit_event** – If True, emit an 'esc_move' event. There are a few cases (ie setting the coordinates after the component is created, but before the entity is added to the terminal) where this is undesirable.

relative_move(*dx*, *dy*, *emit_event=True*)

Move the Entity to a specified position relative to its current position.

Parameters

- **dx** – Movement along X axis, in chars
- **dy** – Movement along Y axis, in chars
- **emit_event** – gets passed to `self.move()` under the hood.

on_event(*event*)

Process tick, if dx != 0 or dy != 0

Parameters **event** – A BearEvent instance

class bear_hug.ecs.**DestructorComponent**(**args*, *is_destroying=False*, ***kwargs*)

Bases: *bear_hug.ecs.Component*

A component responsible for cleanly destroying its entity and everything that has to do with it.

When used, all owner's components except this one are unsubscribed from all events. The deletion does not happen until tick end, to let any running interactions involving the owner finish cleanly.

destroy()

Destroy this component's owner.

Unsubscribes owner and all its components from the queue and sends 'ecs_remove'. Then all components are deleted. Entity itself is left at the mercy of garbage collector.

class bear_hug.ecs.**CollisionComponent**(**args*, *depth=0*, *z_shift=(0, 0)*, *face_position=(0, 0)*, *face_size=(0, 0)*, *passable=False*, ***kwargs*)

Bases: *bear_hug.ecs.Component*

A component responsible for processing collisions of this object.

Stores the following data:

`depth`: Int, a number of additional Z-levels over which collision is possible. Additional collisions are detected on lower Z-levels, ie the level where the object is displayed is always considered to be the front. Defaults to 0, ie collides only to the objects within its own Z-level.

`z_shift`: A 2-tuple of ints. Every next Z-level is offset from the previous one by this much, to create perspective. Defaults to (0, 0), ie no offset.

`face_position`: A tuple of ints describing upper left corner of the collidable part of the entity on the top Z-level. Defaults to (0, 0), ie the upper left corner of the widget is where the hitbox begins. This is a suitable default for flat items, but not for something drawn in perspective.

`face_size`: A tuple of ints describing the size of the collidable part of the entity on the top Z-level. If set to (0, 0), entire entity widget is considered collidable. Defaults to (0, 0). There is no method for making uncollidable entities via setting zero face size; for that, just create your entities without any CollisionComponent at all.

`passable`: whether collisions with this item should be blocking. This class by itself does nothing with this knowledge, but child classes may need it to make distinction between collisions where further movement is impossible (eg walls) and collisions that should be detected, but do not prevent movement (eg walking through fire). Defaults to False, ie blocking collision.

This is a base class, so its event processing just calls `self.collided_into(other_entity)` when owner moves into something, and `self.collided_by(other_entity)` when something else moves into the owner. Both methods do nothing by themselves;actual collision processing logic should be provided by subclasses.

Creating entities with the CollisionComponent but without either PositionComponent or WidgetComponent is just asking for trouble.

class bear_hug.ecs.**WalkerCollisionComponent**(**args*, ***kwargs*)
Bases: *bear_hug.ecs.CollisionComponent*

A collision component that, upon colliding into something impassable (or screen edges), moves the entity back to where it came from.

Expects both entities involved to have a PositionComponent and a PassabilityComponent.

class bear_hug.ecs.**DecayComponent**(**args*, *destroy_condition='keypress'*, *lifetime=1.0*, *age=0*, ***kwargs*)
Bases: *bear_hug.ecs.Component*

Attaches to an entity and destroys it when conditions are met.

Expects the owner to have DestructorComponent.

Parameters

- **destroy_condition** – either 'keypress' or 'timeout'
- **lifetime** – time between entity creation and its destruction. Does nothing if `destroy_condition` is set to 'keypress'. Defaults to 1 second.
- **age** – the age of a given entity. Not meant to be set explicitly, except during deserialization.

class bear_hug.ecs.**CollisionListener**(**args*, ***kwargs*)
Bases: *bear_hug.widgets.Listener*

A listener responsible for detecting collision

bear_hug.ecs.**deserialize_component**(*serial*, *dispatcher*)
Load the component from a JSON string or dict.

Does not subscribe a component to anything (which can be done either by a caller or in the `ComponentClass.__init__`) or assign it to any Entity (which is probably done within

deserialize_entity). The class of a deserialized Component should be imported by the code that calls this function, or someone within its call stack.

If there is a risk that

Parameters

- **serial** – A valid JSON string or a dict produced by deserializing such a string.
- **dispatcher** – A queue passed to the Component.__init__

Returns a Component instance.

bear_hug.ecs.**deserialize_entity**(*serial*, *dispatcher*)

Load the entity from JSON string or dict.

Does not subscribe a new entity to anything or emit bear_create events; this should be done by a caller. All components within the entity are deserialized by calls to deserialize_component

Parameters **serial** – A valid JSON string or a dict produced by deserializing such a string.

Returns an Entity instance

bear_hug.ecs_widgets module

Two Layouts designed specifically for the ECS system.

class bear_hug.ecs_widgets.**ECSLayout**(*chars*, *colors*)

Bases: *bear_hug.widgets.Layout*

A Layout of entities.

This layout, besides visualization, provides collision detection. It is controlled entirely by events. Although Layout methods add_child and move_child are not overloaded, their use is discouraged. Just like a regular Layout, ECSLayout resides within a single bearlibterminal layer and therefore does not provide character overlap for overlapping entities. Who hides whom is currently determined by the order of widget addition, with newer entities on top of older ones (Z-levels are to be implemented in a future release).

Event conventions are as following:

```
BearEvent(event_type='ecs_create', event_value=entity_object)
```

Announces that a new Entity has been created and needs to be registered for ECSLayout. Does not cause it to be placed on screen. The same event tells the EntityTracker about any newly-created Entities. It should not be emitted until the Entity has at least an ID and a WidgetComponent.

```
BearEvent(event_type='ecs_add', event_value=(entity, x, y)).
```

Announces that the widget of the entity in question should be added to the ECSLayout at (x;y). This event should not be emitted before both entity and its widget have been created, and 'ecs_create' event has been emitted.

```
BearEvent(event_type='ecs_move', event_value=(entity, x, y))
```

Announces that the widget of the entity in question should be moved to (x; y). If the widget collides into the widget of another Entity (or multiple widgets), emits BearEvent('ecs_collide', other_entity_id) for each Entity that was collided into. If the widget touches Layout edges, emits BearEvent('ecs_collide', None) instead. In either case, collision does not automatically prevent movement.

```
BearEvent(event_type='ecs_remove', event_value=entity)
```

Announces that the widget of a given entity should be removed from the ECSLayout, but does not cause or imply the its destruction. It is to be used when the Entity currently on screen needs to be hidden, but is expected to be shown again later.

`BearEvent(event_type='ecs_destroy', event_value=entity)`

Announces that the widget of a given entity should be removed from the ECSLayout, as well as from its entities and widgets lists. This event is emitted when the entity is destroyed (eg by DestructorComponent) and used by EntityTracker to know which Entities no longer exist.

`BearEvent(event_type='ecs_redraw')`

Announces that the layout needs to be redrawn this tick, even if none of the events above have been emitted. This is useful if some widget (eg animation) has changed its chars or colors, but was not moved, added or deleted.

If at least one of these events was sent to the ECSLayout, it will redraw itself on 'tick_over'.

Parameters

- **chars** – Layout BG chars
- **colors** – Layout BG colors

add_entity(*entity*)
Register the entity to be displayed.

Assumes that the entity has a widget already. The widget is not actually shown until the *'ecs_add'* event with its entity ID is emitted.

Parameters **entity** – Entity instance

remove_entity(*entity_id*)
Forget about the registered entity and its widget.

Does not imply or cause the destruction of Entity object itself or any of its Component objects. Making sure that the entity is removed cleanly is not the Layout's job.

Parameters **entity_id** – Entity ID

on_event(*event*)
See class documentation

Parameters **event** – BearEvent instance

class bear_hug.ecs_widgets.**ScrollableECSLayout**(*chars*, *colors*, *view_pos=(0, 0)*, *view_size=(10, 10)*)

Bases: *bear_hug.widgets.Layout*

A ECSLayout that can show only a part of its surface.

Like a ScrollableLayout, accepts `chars` and `colors` on creation, which should be the size of the entire layout, not just the visible area. The latter is initialized by `view_pos` and `view_size` arguments.

This class supports all 'ecs_*' events described in the docs for ECSLayout. In addition, it supports the following two two event types:

`BearEvent(event_type='ecs_scroll_by', event_value=(x, y))`

Shifts visible area by x chars horizontally and by y chars vertically.

`BearEvent(event_type='ecs_scroll_to', event_value=(x, y)`

Moves visible area to (x, y).

Both events cause BearLayoutException if event values require visible area to move beyond Layout borders.

Parameters

- **chars** – Layout BG chars.
- **colors** – Layout BG colors.

- **view_pos** – Top left corner of the initial visible area, 2-tuple (x, y).
- **view_size** – The size of the visible area, 2-tuple (x, y).

add_child(*child*, *pos*, *skip_checks=False*)
Add a widget as a child at a given position.

The child has to be a Widget or a Widget subclass that haven't yet been added to this Layout and whose dimensions are less than or equal to the Layout's. The position is in the Layout coordinates, ie relative to its top left corner.

Parameters

- **child** – A widget to add.
- **pos** – A widget position, (x, y) 2-tuple

resize_view(*new_size*)
Currently not implemented. :param new_size: :return:

scroll_to(*pos*)
Move field of view to *pos*.

Raises *BearLayoutException* on incorrect position

Parameters **pos** – 2-tuple (x, y)

scroll_by(*shift*)
Move field of view by *shift[0]* to the right and by *shift[1]* down.

Raises *BearLayoutException* on incorrect position.

Parameters **shift** – 2-tuple (dx, dy)

add_entity(*entity*)
Register the entity to be displayed. Assumes that the entity has a widget already.

The entity is not actually shown until the *'ecs_add'* event is emitted :return:

remove_entity(*entity_id*)
Forget about the registered entity and its widget. Does not imply or cause the destruction of Entity object or any of its Component objects (except if this was the last reference). Making sure that the entity is removed cleanly is someone else's job.

Parameters **entity_id** – ID of the removed entity.

on_event(*event*)
See class documentation.

Parameters **event** – BearEvent instance.

bear_hug.event module

An event system.

Contains a base event class (`BearEvent`) and a queue.

All events are added to the queue and passed around to listeners' `on_event` methods according to their `event_type` subscriptions. This happens when `dispatcher.dispatch_events()` is called, normally every tick. *on_event* callback may return either nothing, a BearEvent, or a list of BearEvents. If any events are returned, they are added to the queue (preserving the order, if there were multiple events within a single return list).

In order to be processed, an event needs to have a correct `event_type`. Builtin types are the following:

'tick', emitted every tick. `event_value` stores time since the previous such event.

'service', emitted for various events related to the queue or loop functioning. Example event_types are 'tick_over' and 'shutdown', emitted during the end of tick and for shutting down the queue.

'key_down', emitted whenever a key or mouse button is pressed. `event_value` stores TK code for the button.

'key_up', emitted whenever a key or mouse button is released. `event_value` stores TK code for the button.

'misc_input', emitted whenever there is some non-keyboard input, for example mouse movement or game window closed via OS UI. `event_value` stores TK code for the input event.

'text_input', emitted when InputField widget wants to return something. `event_value` stores the user-entered string.

'play_sound', emitted when someone has requested a sound to be played. `event_value` stores the sound ID.

ECS events:

'ecs_create', 'ecs_add', 'ecs_move','ecs_collision', 'ecs_destroy', 'ecs_remove', 'ecs_scroll_by', 'ecs_scroll_to', 'ecs_update'. These are described in detail within bear_hug.ecs_widgets docs.

Any user-defined `event_type` needs to be registered before use via `dispatcher.register_event_type()`. Unknown event types can not be added to the queue. Event values, on the other hand, are not validated at all.

class `bear_hug.event.`**BearEvent**(*event_type='tick'*, *event_value=None*)

Bases: `object`

Event data class.

class `bear_hug.event.`**BearEventDispatcher**

Bases: `object`

The BearEvent queue and dispatcher class.

Stores the events sent to it, then emits them to subscribers in chronological order. To start getting events, a Listener needs to subscribe via `dispatcher.register_listener()`.

register_listener(*listener*, *event_types='all'*)

Add a listener to this event_dispatcher.

Any object with an `on_event` method can be added as a listener. This method should accept BearEvent as a single argument and return either nothing, or a single BearEvent, or a list of BearEvents.

To choose event types to subscribe to, `event_types` kwarg can be set to a string or an iterable of strings. If an iterable, its elements should be event types the listener subscribes to.

If a string, the following rules apply:

1. If a string equals 'all', the listener is subscribed to all currently registered event types.

2. Elif a string starts with '*', the listener is subscribed to all currently registered event types for whose type `event_types[1:]` is a substring (regardless of its position). For example, '*ecs' subscribes to all ECS events, like 'ecs_add', 'ecs_move', 'ecs_remove' and so on; '*move' would subscribe only to 'ecs_move' and 'ecs_remove'.

3. Else a string is interpreted as a single event type.

Whether in list or string, unregistered event types raise BearLoopException.

Parameters

- **listener** – a listener to add.
- **event_types** – event types to which it wants to subscribe

unregister_listener(*listener*, *event_types='all'*)
Unsubscribe a listener from all or some of its event types.

Parameters

- **listener** – listener to unsubscribe
- **event_types** – a list of event types to unsubscribe from or 'all'. Defaults to 'all'

register_event_type(*event_type*)
Add a new event type to be processed by queue.

This makes passing (and subscribing to) a new event type possible. No listeners are automatically subscribed to it, even those that were initially registered with 'all' or fitting '*'-types.

Parameters **event_type** – A string to be used as an event type.

add_event(*event*)
Add a BearEvent to the queue.

Parameters **event** –

Returns

start_queue()
Send the queue initialization event. :return:

dispatch_events()
Dispatch all the events to their listeners.

Whatever they return is added to the queue.

dump_queue()
Remove all events from queue without sending them to their recipients

Returns

bear_hug.resources module

Loaders for the various ASCII-art formats.

class bear_hug.resources.**ASCIILoader**
Bases: object

A base class for all the resource loaders. It knows how to return its chars and colors (or a fragment thereof), but expects the children to do their loading by themselves.

get_image()
Return the entire chars and colors of this loader

Returns chars, colors (two 2-nested lists of equal size)

get_image_region(*x*, *y*, *xsize*, *ysize*)
Return some rectangular region of this loader's chars and colors.

Parameters

- **x** – X coordinate of the leftmost column of required region.
- **y** – Y coordinate of the topmost row of required region.
- **xsize** – width of the required region.
- **ysize** – height of the required region.

Returns chars, colors (two 2-nested lists of equal size).

class bear_hug.resources.**TxtLoader**(*filename*, *default_color='white'*, *load_file=False*)
Bases: *bear_hug.resources.ASCIILoader*

A loader that reads a plaintext file.

Accepts a filename (anything acceptable by *open()*) as a single position argument. Since plaintext files don't store colour data, all loaded chars will be the same colour.

Parameters

- **default_color** – the color of chars.
- **load_file** – if True, the file is parsed immediately. Otherwise, only its existence is checked on loader creation, but the file is not parsed until something is required from this loader.

get_image()
Return the entire chars and colors of this loader :returns: chars, colors (two 2-nested lists of equal size)

get_image_region(*x*, *y*, *xsize*, *ysize*)
Return some rectangular region of this loader's chars and colors.

Parameters

- **x** – X coordinate of the leftmost column of required region.
- **y** – Y coordinate of the topmost row of required region.
- **xsize** – width of the required region.
- **ysize** – height of the required region.

Returns chars, colors (two 2-nested lists of equal size).

class bear_hug.resources.**XpLoader**(*filename*, *default_color='white'*)
Bases: *bear_hug.resources.ASCIILoader*

A loader that reads REXPaint `*.xp` files. The file is never parsed until one of the `get_*` methods gets called. Its existence, though, is checked on Loader creation.

As the bear_hug widget API does not allow multi-layered widgets, `get_image` and `get_image_region` return the image with the only the character and color from the highest non-empty layer. For getting data from layers separately, use `get_layer` and `get_layer_region`.

Background colors are ignored altogether.

Most of the XP parsing in this class code is taken from MIT-licensed XPLoaderPy3, copyright Sean Hagar, Erwan Castioni and Gawein Le Goff.

get_image()
Return chars and colors for the entire image. For each cell only the values from the topmost layer are used. Background colors are ignored altogether.

Returns chars, colors (2 2-nested lists)

get_image_region(*x*, *y*, *xsize*, *ysize*)
Return chars and colors for the image region.

For each cell only the values from the topmost layer are used. Background colors are ignored altogether.

Parameters

- **x** – leftmost column of the required region.
- **y** – topmost row of the required region.
- **xsize** – width of the required region

- **ysize** – height of the required region

Returns chars, colors (2 2-nested lists)

get_layer(*layer*)
Get chars and (foreground) colors for the entire image layer.

This method does not check layer size and returns whatever is available. By default, the REXPaint creates layers the size of the entire image, so this shouldn't be much of an issue.

Parameters **layer** – layer number

Returns chars, colors (2 2-nested lists)

get_layer_region(*layer*, *x*, *y*, *xsize*, *ysize*)
Return a rectangular region from a certain layer.

Parameters

- **layer** – layer number
- **x** – leftmost column of the required region.
- **y** – topmost row of the required region.
- **xsize** – width of the required region
- **ysize** – height of the required region

Returns chars, colors (2 2-nested lists)

class bear_hug.resources.**Atlas**(*loader*, *json_file*)
Bases: object

An image atlas.

An instance of this class accepts a Loader instance and a path to the JSON file. The latter is parsed immediately and should contain a list of objects, each of which has five keys: "name", "x", "y", "xsize" and "ysize". Other keys, if any, are ignored. The purpose of this is, basically, to be able to address image regions by a human-readable name, so coordinates and sizes should describe valid regions in the loader. A single region may be called by multiple names, but not the other way around. Other values for the elements, if any, are ignored.

Parameters

- **loader** – a Loader instance.
- **json_file** – path to a JSON file (str).

get_element(*name*)
Return an element with a given name.

If nothing with this name was described in JSON file, raises KeyError

Parameters **name** – A sub-image ID.

Returns A region (chars, colors)

class bear_hug.resources.**Multiatlas**(*atlases*)
Bases: object

A class that contains multiple atlases and fetches elements from whichever has the element you need.

This class supports the *get_element* method similar to Atlas, but does not inherit from it. Consequently, Multiatlas can be created empty and its atlas list can be extended with valid atlases later.

Parameters **atlases** – an iterable of Atlas instances

add_atlas(*atlas*)
Add an atlas to the multiatlas

Checks whether an atlas is valid and has any names shared with other atlases. If so, throws an exception :param atlas: :return:

bear_hug.sound module

A sound system.

Currently it exports a single class called *SoundListener*. It's a Listener wrapper around `simpleaudio` and `wave` libraries. While later the backend is likely to change (at least to support sound formats other than *.wav*), event API is probably gonna remain backwards-compatible.

class bear_hug.sound.**SoundListener**(*sounds*)
Bases: *bear_hug.widgets.Listener*

It doesn't listen to sounds. It listens to the *events* and plays sounds.

This class is expected to be used as a singleton, ie there is no reason to have two SoundListeners active at the same time, and therefore no API for it.

Accepts a single kind of event:

BearEvent(event_type='play_sound', event_value=sound_name)

If sound_name is a known sound ID, this sound is (asynchronously) played. Otherwise, BearSoundException is raised. Sounds can be either supplied in a single arg during creation, or added later via register_sound. In either case, for a sound either a *simpleaudio.WaveObject* or a string is expected. In the latter case, a string is treated as a path to a *.wav* file.

Parameters **sounds** – a dict of `{'sound_id': simlpleaudio.WaveObject}`

register_sound(*sound*, *sound_name*)
Register a new sound for this listener

Parameters

- **sound** – WaveObject or str. A sound to be registered. If str, this is treated as a path to a .wav file.
- **sound_name** – name of this sound.

play_sound(*sound_name*)
Play a sound.

In case you need to play the sound without requesting it through the event.

Parameters **sound_name** – A sound to play.

bear_hug.widgets module

Various useful Widget and Listener classes These widgets and listeners are usable outside the ECS and should be sufficient for simpler games and apps. However, for the sake of clearer architecture, entities are recommended.

bear_hug.widgets.**deserialize_widget**(*serial*, *atlas=None*)
Provided a JSON string, return a widget it encodes.

Specifics of JSON format are described in the Widget class documentation. It is important to know, though, that the Widget subclass that a given JSON encodes should be imported to the code that attempts to call this function.

Parameters **serial** – a JSON string or dict

Returns a Widget instance

bear_hug.widgets.**deserialize_animation**(*serial*, *atlas=None*)
Deserialize an animation from a JSON dump

Parameters **serial** – A JSON string or a dict.

Returns an Animation instance.

class bear_hug.widgets.**Widget**(*chars*, *colors*, *z_level=0*)
Bases: object

The base class for things that can be placed on the terminal.

This class is inactive and is intended to be either inherited from or used for non-interactive non-animated objects. Event processing and animations are covered by its subclasses; while it has on_event() method, it does nothing. This allows Widgets to work without subscribing to the queue and saves some work on not redrawing them unless the Widget itself considers it necessary.

Under the hood, this class does little more than store two 2-nested lists of chars and colors (for characters that comprise the image and their colors). These two should be exactly the same shape, otherwise a BearException is raised.

Widgets can be serialized into JSON similarly to Components and Entities. *repr(widget)* is used for serialization and should generate a valid JSON-encoded dict. It should always include a class key which should equal the class name for that component and will be used by a deserializer to determine what to create. chars and ‘‘colors‘ keys are also necessary. They should encode widget's chars and colors as arrays of strings and each of these strings should be a list of values for chars' and colors' inner lists (str-converted chars and str-converted *#ffffff*-type colors; comma-separated for colors).

All other keys will be deserialized and treated as kwargs to a newly-created object. To define the deserialization protocol, JSON dict may also contain keys formatted as {kwarg_name}_type’‘ which should be a string and will be eval-ed during deserialization. Only Python's builtin converters (eg str, int or float) are safe; custom ones are currently unsupported.

For example, the following is a valid JSON:

```
{"class": "MyWidget",
"chars": ["b,b,b", "a,b,a", "b,a,b"],
"colors": ["#fff,#fff,#fff", "#000,#fff,#000", "#fff,#000,#fff"],
"former_owners": ["asd", "zxc", "qwe"],
"former_owners_type": "set"}
```

Its deserialization is equivalent to the following call:

```
x = MyWidget(chars=[['bbb'],
                    ['aba'],
                    ['bab']],
             colors=[['#fff','#fff','#fff'],
                     ['#000','#fff','#000'],
                     ['#fff','#000','#fff']],
             former_owners=set(['asd, 'zxc', 'qwe']))
```

The following keys are forbidden: parent and terminal. Kwarg validity is not controlled except by WidgetSubclass.__init__().

Parameters

- **chars** – a 2-nested list of unicode characters

- **colors** – a 2-nested list of colors. Anything that is accepted by `terminal.color()` goes here (a color name or a 0xAARRGGBB/0xRRGGBB/0xRGB/0xARGB integer are fine, (r, g, b) tuples are unreliable).
- **z_level** – a Z-level to determine objects' overlap. Used by (Scrollable)ECSLayout. Not to be mixed up with a terminal layer, these are two independent systems.

flip(*axis*)

Flip a widget along one of the axes.

Note that this method has **extremely** limited uses: first, it only affects chars and colors *as they are now*. If later the widget gets updated via animation, updating label text, Layout's children being redrawn, etc., it will be un-flipped again.

Second, most ASCII-art just doesn't take it well. Unlike raster and vector graphics, there is no general way to flip an ASCII image programmatically (except, of course, flipping chars themselves which I find aesthetically unacceptable for my projects). It may work for random noisy tiles, like littered floors, grass and such, but for complex images it's better to provide both left and right versions.

Parameters **axis** – An axis along which to flip. Either one of {'x', 'horizontal'} or one of {'y', 'vertical'}

Returns

class bear_hug.widgets.**SwitchingWidget**(*chars=None*, *colors=None*, *images_dict=None*, *initial_image=None*, ***kwargs*)

Bases: *bear_hug.widgets.Widget*

A widget that can contain a collection of chars/colors pairs and switch them on command.

These char/color pairs should all be the same shape. Does not do any transition animations.

`chars` and `colors` args, although accepted during creation, are discarded. They do not affect the created widget in any way, nor are they shown at any moment.

Parameters

- **images_dict** – a dict of {image_id: (chars, colors)}
- **initial_image** – an ID of the first image to show. Should be a key in `images_dict`.

switch_to_image(*image_id*)

Switch to a given image ID

The ID should be a key in the original `image_dict`. Otherwise, BearException is raised.

Parameters **image_id** – image ID, str.

class bear_hug.widgets.**Layout**(*chars*, *colors*, ***kwargs*)

Bases: *bear_hug.widgets.Widget*

A widget that can add others as its children.

All children get drawn to its chars and colors, and are thus displayed within a single bearlibterminal layer. Therefore, if children overlap each other, the lower one is hidden completely. In the resolution of who covers whom, a newer child always wins. The layout does not explicitly pass events to its children, they are expected to subscribe to event queue by themselves.

The Layout is initialized with a single child, which is given chars and colors provided at Layout creation. This child is available as `l.children[0]` or as `l.background`. Its type is always `Widget`.

The Layout automatically redraws itself on *tick* event, whether its children have updated or not.

Does not support JSON serialization

Parameters

- **chars** – chars for layout BG.
- **colors** – colors for layout BG.

add_child(*child*, *pos*, *skip_checks=False*)
Add a widget as a child at a given position.

The child has to be a Widget or a Widget subclass that haven't yet been added to this Layout and whose dimensions are less than or equal to the Layout's. The position is in the Layout coordinates, ie relative to its top left corner.

Parameters

- **child** – A widget to add.
- **pos** – A widget position, (x, y) 2-tuple

remove_child(*child*, *remove_completely=True*)
Remove a child from a Layout.

Parameters

- **child** – the child to remove
- **remove_completely** – if False, the child is only removed from the

screen, but remains in the children list. This is not intended to be used and is included only to prevent `self.move_child` from messing with child order.

move_child(*child*, *new_pos*)
Remove the child and add it at a new position.

Parameters

- **child** – A child Widget
- **new_pos** – An (x, y) 2-tuple within the layout.

on_event(*event*)
Redraw itself, if necessary

get_absolute_pos(*relative_pos*)
Get an absolute position (in terminal coordinates) for any location within self.

Parameters **relative_pos** – An (x, y) 2-tuple in Layout coordinates

Returns An (x, y) 2-tuple for the same point in terminal coordinates.

get_child_on_pos(*pos*, *return_bg=False*)
Return the newest child on a given position.

Parameters

- **pos** – Position in Layout coordinates
- **return_bg** – If True, return background widget when clicking outside any children. If False, return None in this case. Defaults to False

Returns Widget instance or None

class bear_hug.widgets.**ScrollBar**(*orientation='vertical'*, *length=10*, *colors=('gray'*, *'white')*, ***kwargs*)
Bases: *bear_hug.widgets.Widget*

A scrollbar to be used with ScrollableLayout.

Does not accept input, does not support serialization.

Parameters

- **orientation** – Scrolling direction. One of 'vertical' or 'horizontal'
- **length** – Scrollbar length, in chars.
- **colors** – A 2-tuple of (BG colour, moving thingy colour)

show_pos(*position*, *percentage*)
Move the scrollbar.

Parameters **position** – Float. The position of the top (or left) side of the

scrollbar, as part of its length

Parameters **percentage** – Float. The lengths of the scrollbar, as part of the

total bar length

class bear_hug.widgets.**ScrollableLayout**(*chars*, *colors*, *view_pos=(0, 0)*, *view_size=(10, 10)*, ***kwargs*)

Bases: *bear_hug.widgets.Layout*

A Layout that can show only a part of its surface.

Like a Layout, accepts *chars* and *colors* on creation, which should be the size of the entire layout, not the visible area. The latter is initialized by *view_pos* and *view_size* arguments.

Does not support JSON serialization.

Parameters

- **chars** – Layout BG chars.
- **colors** – Layout BG colors.
- **view_pos** – a 2-tuple (x,y) for the top left corner of visible area, in Layout coordinates.
- **view_size** – a 2-tuple (width, height) for the size of visible area.

scroll_to(*pos*)
Move field of view to `pos`.

Raises `BearLayoutException` on incorrect position

Parameters **pos** – A 2-tuple of (x, y) in layout coordinates

scroll_by(*shift*)
Move field of view by `shift[0]` to the right and by `shift[1]` down.

Raises `BearLayoutException` on incorrect position

Parameters **shift** – A 2-tuple of (dx, dy) in layout coordinates

class bear_hug.widgets.**InputScrollable**(*chars*, *colors*, *view_pos=(0, 0)*, *view_size=(10, 10)*, *bottom_bar=False*, *right_bar=False*, ***kwargs*)

Bases: *bear_hug.widgets.Layout*

A ScrollableLayout wrapper that accepts input events and supports the usual scrollable view bells and whistles. Like ScrollableLayout, accepts chars and colors the size of the *entire* layout and inits visible area using view_pos and view_size.

If bottom_bar and/or right_bar is set to True, it will be made one char bigger than view_size in the corresponding dimension to add ScrollBar.

Can be scrolled by arrow keys.

Does not support JSON serialization

class bear_hug.widgets.**Animation**(*frames*, *fps*, *frame_ids=None*)
Bases: object

A data class for animation, *ie* the sequence of the frames

Animation can be serialized to JSON, preserving fps and either frame dumps (similarly to widget chars and colors) or frame image IDs. For the latter to work, these IDs should be provided during Animation creation via an optional `frame_ids` kwarg. The deserializer will then use them with whichever atlas is supplied to create the animation.

Since this class has no idea of atlases and is unaware whether it was created with the same atlas as deserializer will use (which REALLY should be the same, doing otherwise is just asking for trouble), frame ID validity is not checked until deserialization and, if incorrect, are not guaranteed to work.

Parameters

- **frames** – a list of (chars, colors) tuples
- **fps** – animation speed, in frames per second. If higher than terminal FPS, animation will be shown at terminal FPS.
- **frame_ids** – an optional list of frame names in atlas, to avoid dumping frames. Raises `BearJSONException` if its length isn't equal to that of frames.

class bear_hug.widgets.**SimpleAnimationWidget**(*animation*, **args*, *is_running=True*, *emit_ecs=True*, *z_level=0*)
Bases: *bear_hug.widgets.Widget*

A simple animated widget that cycles through the frames.

Parameters

- **frames** – An iterable of (chars, colors) tuples. These should all be the same size.
- **fps** – Animation speed, in frames per second. If higher than terminal FPS, it will be slowed down.
- **emit_ecs** – If True, emit ecs_update events on every frame. Useless for widgets outside ECS, but those on `ECSLayout` are not redrawn unless this event is emitted or something else causes ECSLayout to redraw.

class bear_hug.widgets.**MultipleAnimationWidget**(*animations*, *initial_animation*, *emit_ecs=True*, *cycle=False*, *z_level=0*)
Bases: *bear_hug.widgets.Widget*

A widget that is able to display multiple animations.

Plays only one of the animations, unless ordered to change it by `self.set_animation()`

Parameters

- **animations** – A dict of `{animation_id: Animation()}`
- **initial_animation** – the animation to start from.
- **emit_ecs** – If True, emit ecs_update events on every frame. Useless for widgets outside ECS, but those on `ECSLayout` are not redrawn unless this event is emitted or something else causes the layout to redraw.
- **cycle** – if True, cycles the animation indefinitely. Otherwise stops at the last frame.

set_animation(*anim_id*, *cycle=False*)
Set the next animation to be played.

Parameters

- **anim_id** – Animation ID. Should be present in self.animations
- **cycle** – Whether to cycle the animation. Default False.

class bear_hug.widgets.**Label**(*text*, *chars=None*, *colors=None*, *just='left'*, *color='white'*, *width=None*, *height=None*, ***kwargs*)

Bases: *bear_hug.widgets.Widget*

A widget that displays text.

Accepts only a single string, whether single- or multiline (ie containing ‘‘

‘‘ or not). Does not support any complex text markup. Label’s text can be edited at any time by setting label.text property. Note that it overwrites any changes to `self.chars` and `self.colors` made after setting `self.text` the last time.

Unlike text, Label’s height and width cannot be changed. Set these to accomodate all possible inputs during Label creation. If a text is too big to fit into the Label, ValueError is raised.

param text string to be displayed

param just horizontal text justification, one of ‘left’, ‘right’

or ‘center’. Default ‘left’.

param color bearlibterminal-compatible color. Default ‘white’

param width text area width. Defaults to the length of the longest ‘‘

`-delimited substring in ``text.`

param height text area height. Defaults to the line count in *text*

class bear_hug.widgets.**InputField**(*name='Input field'*, *accept_input=True*, *finishing=False*, ***kwargs*)

Bases: *bear_hug.widgets.Label*

A single-line field for keyboard input.

The length of the input line is limited by the InputField size. When the input is finished (by pressing ENTER), InputField emits a `BearEvent(event_type='text_input', event_value=(field.name, field.text))`

Since BLT has no support for system keyboard layouts, only supports QWERTY Latin. This also applies to non-letter symbols: for example, comma and period are considered to be different keys even in Russian layout, where they are on the same physical key.

finish()

Finish accepting the input and emit the ‘text_input’ event at the next opportunity. This opportunity will not present itself until the next event is passed to `self.on_event`.

class bear_hug.widgets.**MenuWidget**(*dispatcher*, *terminal=None*, *items=[]*, *header=None*, *color='white'*, *items_pos=(2, 2)*, *background=None*, *switch_sound=None*, *activation_sound=None*, ***kwargs*)

Bases: *bear_hug.widgets.Layout*

A menu widget that includes multiple buttons.

Parameters

- **dispatcher** – BearEventDispatcher instance to which the menu will subscribe
- **items** – an iterable of MenuItems

- **background** – A background widget for the menu. If not supplied, a default double-thickness box is used. If background widget needs to get events (ie for animation), it should be subscribed by the time it's passed here.
- **color** – A bearlibterminal-compatible color. Used for a menu frame and header text
- **items_pos** – A 2-tuple of ints. A position of top-left corner of the 1st MenuItem
- **header** – str or None. A menu header. This should not be longer than menu width, otherwise an exception is thrown. Header may look ugly with custom backgrounds, since it's only intended for non-custom menus.
- **switch_sound** – str. A sound which should be played (via `play_sound` BearEvent) when a button is highlighted.
- **activation_sound** – str. A sound which should be played (vai `play_sound` BearEvent) when a button is pressed

class bear_hug.widgets.**MenuItem**(*text='Test'*, *action=<function MenuItem.<lambda>>*, *color='white'*, *highlight_color='green'*, ***kwargs*)

Bases: *bear_hug.widgets.Layout*

A button for use inside menus. Includes a label surrounded by a single-width box. Contains a single callable, `self.action`, which will be called when this button is activated.

MenuItem by itself does not handle any input. It provides `self.activate` method which should be called by something (presumably a menu containing this button).

Parameters

- **text** – str. A button label
- **action** – callable. An action that this MenuItem performs. This should return either None, BearEvent or an iterable of BearEvents
- **color** – a bearlibterminal-compatible color that this button has by

default

Parameters **highlight_color** – a bearlibterminal-compatible color that this button

has when highlighted via keyboard menu choice or mouse hover.

highlight()
Change button colors to show that it's highlighted

unhighlight()
Change button colors to show that it's no longer highlighted :return:

activate()
Perform the button's action

class bear_hug.widgets.**FPSCounter**(***kwargs*)

Bases: *bear_hug.widgets.Label*

A simple widget that measures FPS.

Actually just prints 1/(average runtime over the last 100 ticks in seconds), so it takes 100 ticks to get an accurate reading. Not relevant except on the first several seconds of the program run or after FPS has changed, but if it seems like the game takes a second or two to reach the target FPS – it just seems that way.

class bear_hug.widgets.**MousePosWidget**(***kwargs*)

Bases: *bear_hug.widgets.Label*

A simple widget that reports current mouse position.

In order to work, it needs `self.terminal` to be set to the current terminal, which means it should either be added to the terminal directly (without any Layouts) or terminal should be set manually before MousePosWidget gets its first `tick` event. It is also important that this class uses `misc_input:TK_MOUSE_MOVE` events to determine mouse position, so it would report a default value of '000x000' until the mouse has moved at least once.

class bear_hug.widgets.**Listener**(*terminal=None*)
Bases: object

A base class for the things that need to interact with the queue (and maybe the terminal), but aren't Widgets.

Parameters **terminal** – BearTerminal instance

on_event(*event*)
The event callback. This should be overridden by child classes.

Parameters **event** – BearEvent instance

register_terminal(*terminal*)
Register a terminal with which this listener will interact

Parameters **terminal** – A BearTerminal instance

class bear_hug.widgets.**ClosingListener**
Bases: *bear_hug.widgets.Listener*

The listener that waits for a `TK_CLOSE` input event (Alt-F4 or closing window) and sends the shutdown service event to the queue when it gets one.

All widgets are expected to listen to it and immediately save their data or do whatever they need to do about it. On the next tick ClosingListener closes both terminal and queue altogether.

class bear_hug.widgets.**LoggingListener**(*handle*)
Bases: *bear_hug.widgets.Listener*

A listener that logs the events it gets.

It just prints whatever events it gets to sys.stderr. The correct way to use this class is to subscribe an instance to the events of interest and watch the output. If logging non-builtin events, make sure that their `event_value` can be converted to a string. Converstion uses `str(value)`, not `repr(value)` to avoid dumping entire JSON representations.

1.1.2 Module contents

bear_hug is a library for building ASCII-art games and apps in Python 3.6+.

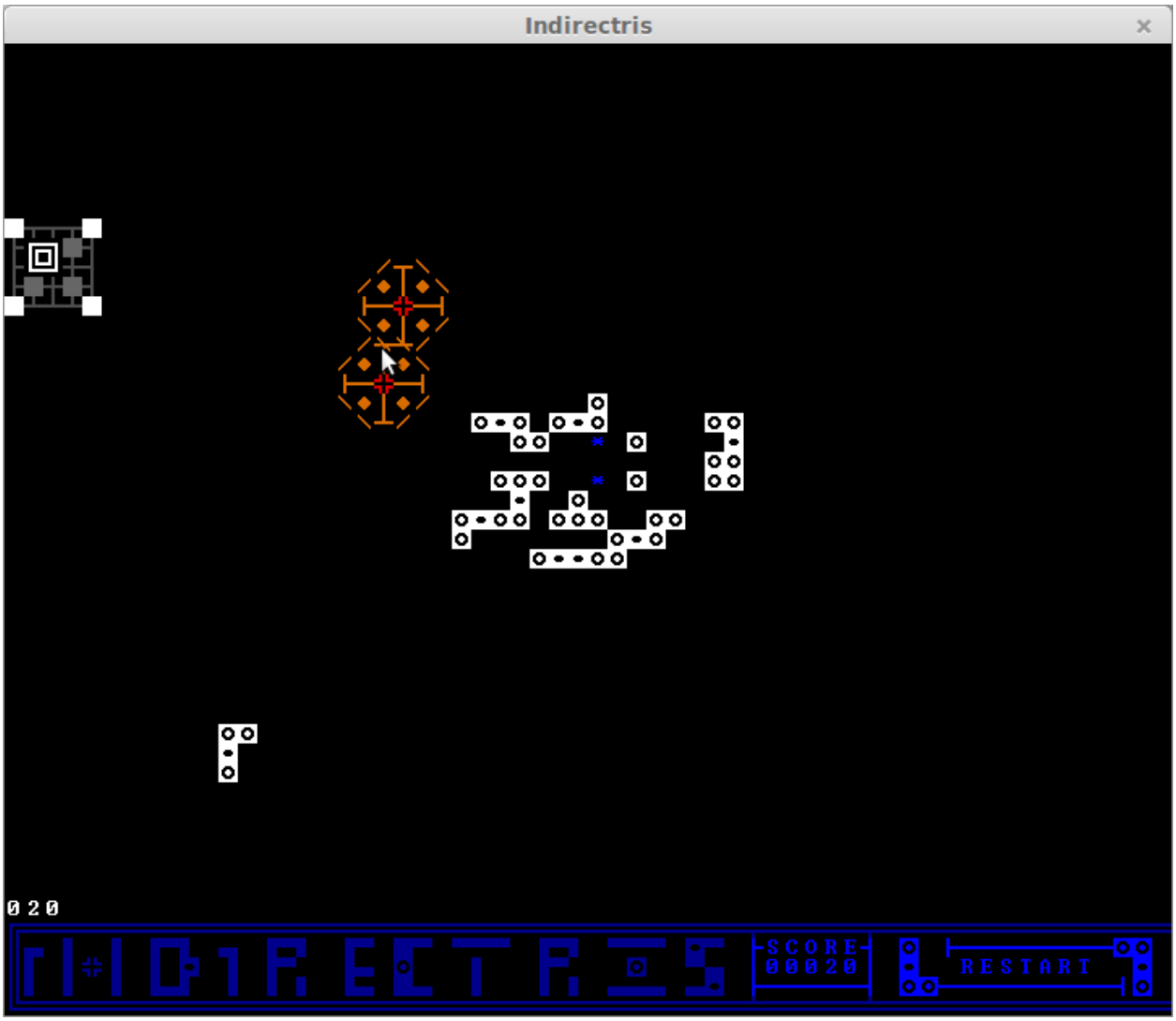
Note that this is *not* curses-like library; bear_hug uses bearlibterminal as a backend, which in turn uses SDL. It is not meant to work in a TTY.

Currently available:

- multiple useful widgets like layouts, labels, animated widgets and so on
- event system
- basic input (keyboard and mouse)
- WAV sound using simpleaudio
- entity-component system for your gamedev needs

Source code is available (and pull requests are welcome) at the github repository

For an example of a simple game made with this library, check out my Ludum Dare 41 gravity-controlled Tetris variant.



(LD page | repository)

This one is made using only basic widgets and events. For a more complex ECS-based game, take a look at this repository (work in progress).

CHAPTER 2

Installation

Stable version can be downloaded from PyPI with

pip install bear_hug

Latest versions can always be downloaded from the repository. Prerequisites are `bearlibterminal` and, if you plan to use sound, `simpleaudio`.

CHAPTER 3

Authors

Alexey Morozov *aka* synedraacus

CHAPTER 4

License

The library itself is available under the terms of MIT license.

Sounds and images included in the demos are available under the terms of CC-BY 3.0

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

F

G

H

I

L

M

O

P

R

S

T

U

V

W

X