kelpy Documentation

Release 0.1

Amanda Yung

September 27, 2016

Contents

The basic approach of Kelpy is to handle simple animations and things, taking care of the main refresh-display loop in pygame, and letting us just construct sequences of object actions and handle events. Future work will handle fancier counterbalancing, etc.

Kelpy is a work in progress, so please be mindful that some demos and classes may not function as intended just yet!

CHAPTER 1

Contents

1.1 Installation

Put this library somewhere–mine lives in:

```
/home/piantado/mit/Libraries/kelpy/
```

Set the PYTHONPATH environment variable to point to the library:

```
export PYTHONPATH=$PYTHONPATH:/home/piantado/Desktop/mit/Libraries/kelpy
```

You can put this into your .bashrc file to make it load automatically when you open a terminal. On ubuntu and most linux, this is:

```
echo 'export PYTHONPATH=$PYTHONPATH:/home/piantado/Desktop/mit/Libraries/kelpy' >> ~/.bashrc

PYTHONPATH="${PYTHONPATH}:/path/to/some/cool/python/package/:/path/to/another/cool/python/package/"
export PYTHONPATH
```

1.1.1 Dependencies

- PIL
- Pygame
- Tobii SDK (optional – for Tobii eye tracker integration)

Tobii SDK Installation The Tobii SDK can be downloaded at their website. In the zip folder, extract the version that matches your OS (including if it's 32 or 64-bit), rename the folder to tobiisdk, and place it in the same parent directory as the kelpy library. So sticking with our example, it would be in:

/home/piantado/mit/Libraries/tobiisdk/

Add the following path to your PYTHONPATH environment variable (as you did previously for adding kelpy):

export PYTHONPATH=$PYTHONPATH:/home/piantado/Desktop/mit/Libraries/tobiisdk/python27/modules

For Linux users, you will also need to install the following additional python modules(using apt-get install):

- libssh2-1
- libavahi-client3

See the Tobii SDK Developers Guide that came with the SDK for more information, if curious.

1.2 Getting Started

Kelpy experiments follow a basic structure:

1. Create the kelpy window via `initialize_kelpy`.
2. Load the stimuli (images, sounds) and add images to the `OrderedUpdates` list.
3. Add animations to the `DisplayQueue`.
4. Run the `kelpy_standard_event_loop` and check for task/user events.

You should always include the following kelpy modules:

- `kelpy.OrderedUpdates`
- `kelpy.DisplayQueue`
- `kelpy.EventHandler`

1.2.1 Key Components

initialize_kelpy()

This sets up the kelpy screen based on the given dimensions and background color. All the stimuli are displayed within this window. For example, this function can be called like so:

```
screen, spots = initialize_kelpy( dimensions=(1024,768), bg=(250,250,250) )
```

Where `screen` is the reference to the screen object and `spots` is a reference to the `StandardLocations` object (see *StandardLocations*).

Alternatively, you can also set the kelpy screen to fullscreen:

```
screen, spots = initialize_kelpy( fullscreen=True, bg=(250,250,250) )
```

OrderedUpdates

OrderedUpdates takes a list of kelpy sprites (e.g., *CommandableImageSprite*) and uses the list order as the order to draw/update the sprites. This order is important when stimuli overlap, such as when needing an occluder to hide an image:

```
# create your kelpy sprites
occluder = CommandableImageSprite(screen, start_position, image_paths[0])
reward = CommandableImageSprite(screen, start_position, image_paths[1])

# add sprites to a list
stimuli = [reward, occluder]

# store the order that we will draw and update things in this variable 'dos'
dos = OrderedUpdates(stimuli)
```

In this example, `reward` is drawn first since it is at [0]. `occluder` is then drawn second – since the two sprites share the same location (`start_position`), `occluder` is drawn on top of `reward`.

For the kelpy screen to recognize this `OrderedUpdates` list, the `dos` object must be passed to the `kelpy_standard_event_loop` (see *kelpy_standard_event_loop()*).

DisplayQueue

The DisplayQueue stores the order of stimulus events that will occur during the `kelpy_standard_event_loop`. Kelpy sprites have certain actions they can perform, such as moving or rotating, which can be executed in the order specified in the DisplayQueue:

```
# create the DisplayQueue
Q = DisplayQueue()

# add the sprite events to the queue:
# first the occluder moves to the north position off-screen (taking 1.5 seconds)
Q.append(obj=occluder, action='move', pos=spots.north, duration=1.5)
# once the occluder is finished, the the reward sprite wiggles for 2 seconds
Q.append(obj=reward, action='wiggle', duration=2, amount=45, cycles=5)
```

You can also set actions to occur simultaneously by using `append_simultaneous()`:

```
# create the DisplayQueue
Q = DisplayQueue()

# have the occluder move while the reward wiggles
Q.append(obj=occluder, action='move', pos=spots.north, duration=1.5)
Q.append_simultaneous(obj=reward, action='wiggle', duration=2, amount=45, cycles=5)
```

Like `OrderedUpdates`, a `DisplayQueue` must be passed to the `kelpy_standard_event_loop`. For a list of specific actions for the different types of kelpy sprites, see *Stimulus Types*.

kelpy_standard_event_loop()

With your kelpy screen, `OrderedUpdates` object, and `DisplayQueue` object, you can now run an event loop:

```
# pass your 3 objects to the event loop.
# the loop will run indefinitely while processing interactions,
# unless you tell it otherwise within the for loop.
for event in kelpy_standard_event_loop(screen, Q, dos):
# in this example, once the reward is clicked, a sound is played
# and then it breaks out of the event loop
        if is_click(event):
                # check what was clicked
                who = who_was_clicked(dos)
                if who is reward:
                        play_sound(sound_file, wait=True, volume=0.7)
                        break
```

Within the for loop, you can detect user events (such as clicking on objects, dragging objects), add additional sprite actions to the `DisplayQueue`, play sounds, and whatever else necessary for the flow of your experiment.

1.2.2 Other Useful Classes/Functions

StandardLocations

Once a kelpy screen is created, a `StandardLocations` object is returned along with a reference to the screen object itself. `StandardLocations` has several pre-defined properties that refer to spots both on and off the screen, based on the dimensions of the kelpy window. These spots can be used for image placement. Description of available positions:

Off-screen locations

- north
- northeast
- east
- southeast
- south
- southwest
- west
- northwest

On-screen locations

Default on-screen locations are based on a 4x4 grid pattern, with rows as letters a-d and columns as numbers 1-4. For example, `a1` is the top left corner, while `d1` is the bottom left corner. There is also a row of 4 positions at the middle height of the screen (`midrow1`, `midrow2`, etc.) and a column of 4 positions at the middle width (`midcol1`, `midcol2`, etc.). The center position of the screen is `center`.

who_was_clicked()

During the `kelpy_standard_event_loop`, if you make a call to `who_was_clicked()` and pass your DisplayQueue as an argument, the function will return what kelpy sprite was clicked on, if any.

```
who = who_was_clicked(dos)
if who is correct_sprite:
        #return something positive
elif who is wrong_sprite:
        #return something negative?
else:
        #possibly do nothing since no sprite was clicked
```

kstimulus()

This function simplifies calling stimuli files that are included with kelpy, located in the `kelpy/stimuli` directory. For example, if you wanted to use the `Beep.wav` sound file:

```
beep = kstimulus('sounds/Beep.wav')
play_sound(beep)
```

There are many images and sounds in the stimuli library, so take some time to browse through them!

filename()

As the function name implies, this returns just the filename (e.g., `image.png`) of the filepath that is passed to it (e.g., `/home/user/kelpy-master/kelpy/stimuli/image.png`). This is useful when outputting either to the command window or to a data file what stimuli were displayed/played.

Q.is_empty()

This is a call to the DisplayQueue to check if it still has anything in its queue. If it does, it returns False. Otherwise, it returns True. This is potentially useful when determining when to move on during a task.

1.2.3 Code Reference

1.3 Stimulus Types

There are several different kinds of stimuli in kelpy. Here is an overview of the types:

1.3.1 Sprites

Sprites are an image that can be manipulated either by the code and/or the user, depending on the type of sprite. Each type of sprite has its own set of actions which are executed using the `DisplayQueue`'s `append()`. All sprites extend the parent class `CommandableSprite`, which has a basic set of actions inherited by all other sprites:

CommandableSprite actions:

- **move** - Moves the sprite linearly from its current location to the specified location in a given amount of time.

  Properties:

  - *duration* - Seconds to take until at the new position. Default is 0 (so movement is instantaneous).
  - *pos* - A tuple containing the (x,y) coordinates of the new position

- **wait** - Simply waits in the current position.

  Properties:

  - *duration* - Seconds to wait. If no other command action is given afterwards, the sprite will continue to wait indefinitely. Default is 0.

- **hide** - Immediately hides the sprite.

  Properties:

  - *duration* - Seconds to wait. This does not specifically control how long the sprite is hidden until it appears again. In order to make the sprite visible again, use the command `show`.

- **show** - Immediately shows the sprite if it is not already visible.

  Properties:

  - *duration* - Seconds to wait. This does not specifically control how long the sprite is visible until it is hidden. In order to stop drawing the sprite, use the command `hide`.

CommandableImageSprite

Note: Requires module `kelpy.CommandableImageSprite`

A `CommandableImageSprite` displays an image that can be manipulated by translation, rotation, and scaling.

```
sprite = CommandableImageSprite(screen, init_position, imagepath, rotation=0, scale=1.0, brightness=1
```

Arguments:

- **screen** A reference to the kelpy screen object.
- **init_position** A tuple that contains the (x,y) coordinates for the initial position of the sprite.
- **imagepath** The filepath of the image to be displayed.

- **rotation** The value of the initial rotation of the sprite. Default is 0.
- **scale** The scale of the image to be used, from 0 to 1 (or beyond to enlarge, beware pixelation!). Default is 1.
- **brightness** Initial brightness of the image. Default is 1.

Returns:

- **sprite** A reference to the created `CommandableImageSprite` object

CommandableImageSprite actions

(Inherits all actions from ''CommandableSprite'', plus...)

- **waggle** - Jiggles the image for a given amount of time. The animation looks like a swing upwards to the right and then back down to its original position.

  Properties:

  - *duration* - The duration of the waggle motion in seconds.
  - *amount* - The amount of rotation (units: ???).
  - *period* - A number from 0-1 that indicates the proportion of the duration that is the length of a cycle (where a cycle is rotating counterclockwise for the given amount and then back to the original position). For example, if the period is 1, then it will go through one cycle; if 0.5, then it will go through 2 cycles. Note: the image will automatically go back to its original position after the duration time has passed (resulting in jerky motion if there are not whole cycles).

- **wagglemove** - A `waggle` action in combination with a `move` action

  Properties:

  - *duration* - The duration of the waggle-move motion in seconds.
  - *amount* - The amount of rotation (units: ???).
  - *period* - The length of a cycle in seconds (where a cycle is rotating counterclockwise for the given amount and then back to the original position). Note: if the duration is not a multiple of the period, the image will automatically go back to its original position after the duration time has passed (resulting in jerky motion).
  - *pos* - A tuple containing the (x,y) coordinates of the new position for the sprite.

- **wiggle** - A modification of the `waggle` action. This motion is oscillatory (moves to the right and then to the left past the original position, like a pendulum).

  Properties:

  - *duration* - The duration of the wiggle motion in seconds.
  - *amount* - The maximum amount of rotation in degrees (i.e., the amplitude).
  - *cycles* - The number of cycles completed in the given duration. Can also use half cycles. Like the waggle action, the sprite will always end at its original position.

- **circlescale** - This grows and shrinks the sprite repeatedly for a given duration.

  Properties:

  - *duration* - The duration of the circlescale animation.
  - *amount* - The amount of the scaling (units: ???)

  - *period* - A number from 0-1 that indicates the proportion of the duration that is the length of a cycle (where a cycle is growing and then shrinking back to its original size). For example, if the period is 1, then it will go through one cycle; if 0.5, then it will go through 2 cycles. Note: the image will automatically go back to its original size after the duration time has passed (resulting in jerky motion if there are not whole cycles).

- **scale** - This scales the image by the given number over an optionally specified time.

  Properties:

  - *duration* - The time it takes to scale the image, in seconds.
  - *amount* - The amount to scale the image. For example, 0.5 shrinks the image to half its size; 2 would double its size.

- **swap** - Swaps the current image for a different image.

  Properties:

  - *image* - The filepath for the new image.
  - *rotation* - The initial rotation of the new image.
  - *scale* - The initial scale of the new image.

- **rotate** - Rotates the image.

  Properties:

  - *amount* - The degrees of rotation.
  - *duration* - The time it takes to rotate the image for the given amount.

- **darken** - Fades the color of the image.

  Properties:

  - *amount* - A number between 0-1 to indicate how faded the image should be (0 makes the image disappear).
  - *duration* - The time it takes to fade the image, in seconds.

- **swapblink** - Swap between two images repeatedly.

  Properties:

  - *duration* - The duration of the swapping animation in seconds.
  - *period* - The duration an image is displayed on the screen before it is swapped in seconds.
  - *image* - The filepath for the new image.
  - *rotation* - The rotation value of the new image in degrees.
  - *scale* - The scale value of the new image.
  - *brightness* - The brightness value of the new image.

- **restore** - Return the sprite to the original image it was created with.

  (No properties necessary)

DragSprite

Note: Requires module `kelpy.DragDrop`

DragSprite is a subtype of CommandableImageSprite. It can be dragged by the user when it is clicked/touched. It is created with the same arguments as a CommandableImageSprite and also inherits the same command actions. In order to use a DragSprite, you must also call process_dragndrop() in your kelpy event loop:

```
# create the DragSprite
draggy = DragSprite(screen, init_position, imagepath)

# add the sprite to an OrderedUpdates object
dos = OrderedUpdates(draggy)

# make the DisplayQueue (no need to add any actions)
Q = DisplayQueue()

# run the kelpy event loop
for event in kelpy_standard_event_loop(screen, Q, dos):
        #check for any drag events and update the sprite's position
        draggy.process_dragndrop(event)
```

When using other sprites with a DragSprite, it is useful to employ bring_clicked_to_top(). This ensures the currently dragged sprite will be drawn on top of other sprites.

```
draggy1 = DragSprite(screen, init_position[0], imagepath[0])
draggy2 = DragSprite(screen, init_position[1], imagepath[1])
draggy3 = DragSprite(screen, init_position[2], imagepath[2])

# create a list of the sprites for easy reference later
sprites = [draggy1, draggy2, draggy3]
dos = OrderedUpdates(sprites)
Q = DisplayQueue()

for event in kelpy_standard_event_loop(screen, Q, dos):
        # use reversed() in order to get the sprite on top
        # (objects first in the array are drawn first,
        # and thus under the other objects)
        for sprite in reversed(sprites):
                if sprite.process_dragndrop(event):
                        bring_clicked_to_top(sprite, sprites, dos)
                        break
```

DropSprite

Note: Requires module kelpy.DragDrop

DropSprite is also a subtype of CommandableImageSprite and should be used in conjunction with DragSprite. They can detect when DragSprites are dropped on top of them.

Like a DragSprite, a DropSprite is created with the same arguments as a CommandableImageSprite. To use DropSprites, you must also call register_drop_zone() for your DragSprites in addition to was_dropped_into_zone() in the kelpy event loop. Check which sprite is on a DropSprite with who_was_dropped().

```
draggy1 = DragSprite(screen, init_position[0], imagepath[0])
draggy2 = DragSprite(screen, init_position[1], imagepath[1])
drop_zone = DropSprite(screen, init_position[2], imagepath[2])
```

```
# register drop zone for the DragSprites
draggy1.register_drop_zone(drop_zone)
draggy2.register_drop_zone(drop_zone)

sprites = [draggy1, draggy2, drop_zone]
dos = OrderedUpdates(sprites)
Q = DisplayQueue()

for event in kelpy_standard_event_loop(screen, Q, dos):
        for sprite in reversed(sprites):
                if sprite.process_dragndrop(event):
                        bring_clicked_to_top(sprite, sprites, dos)
                        break

        # check if a sprite was dropped into a registered drop zone
        if was_dropped_into_zone(event):
                # optionally can check who was dropped
                who = who_was_dropped(event)
                if who is draggy1:
                        # do something
                else:
                        # do something else
```

TextSprite

Note: Requires module `kelpy.TextSprite`

`TextSprite` is for displaying text in the kelpy screen. It only inherits actions from `CommandableSprite`.

```
text = TextSprite(text, screen, init_position)
```

Arguments:

- **text** The string of text to be displayed.
- **screen** A reference to the kelpy screen object.
- **init_position** A tuple that contains the (x,y) coordinates for the initial position of the text (the text is automatically centered at this point).

1.3.2 Other Visuals

AttentionGetter

Note: Requires module `kelpy.AttentionGetter`

An `AttentionGetter` displays an animated .gif image and optionally plays sound simultaneously. This is particularly useful when running infant studies in order to get the participant's attention back to the screen. Unlike the kelpy sprites, `AttentionGetters` are not manipulated within the kelpy event loop.

```
#just call the function and you're done!
gif_attention_getter(screen, position, images, sounds=None, keypress=None, stop_music=True, backgrou
```

Arguments:

- **screen** A reference to the kelpy screen object.
- **position** A tuple that contains the (x,y) coordinates for the initial position of the text (the text is automatically centered at this point).
- **images** Can either be one .gif image filepath or a list of .gif image filepaths to use for the attention getter. If it is a list, the .gif used is selected randomly.
- **sounds** Can either be one sound filepath or a list of sound filepaths. If it is a list, then the sound used is selected randomly. Default is no sound.
- **keypress** A pygame keypress code that triggers the attention getter to stop and disappear. Default is no keypress reaction. See the list of key code constants: http://www.pygame.org/docs/ref/key.html
- **stop_music** If True, the music stops after the .gif is played. Otherwise if False, the music continues to play even after the .gif ends. Default is True.
- **background_color** RGB color of the background that the .gif is displayed on. Default is white.
- **duration** The duration in seconds the .gif plays for (unless there is a keypress which stops it earlier). Default is 4 seconds.

1.3.3 Sounds

Note: No additional kelpy module required.

Sounds can be played in two ways. One is using `play_sound()`, which plays the sound immediately:

```
play_sound(filepath, wait=False, volume=0.65)
```

Arguments:

- **filepath** The filepath for the sound file to be played.
- **wait** If True, the program will wait until the sound is finished playing before it continues. Default is False.
- **volume** A number from 0-1 that sets the sound's volume level. Default is 0.65.

The other option is adding the sound to the DisplayQueue, where it will be played when it comes up in the queue:

```
for event in kelpy_standard_event_loop(screen, Q, dos):

        # have a sprite do something first
        Q.append(obj=sprite, action="rotate", amount=15, duration=1.5)
        # then play sound after the animation is completed
        Q.append(obj='sound', file=filepath, volume=0.5)
```

Arguments:

- **obj** This should be set to `'sound'`, rather than a reference to a sprite object like typical `Q.append` cases.
- **file** The filepath for the sound file to be played.
- **volume** The volume to set for the sound file. Default is 0.5.

1.4 Tobii Integration

Kelpy also supports the use of a Tobii eye tracker in experiments.

Note: As listed in the Tobii SDK Developer's Guide, the following Tobii Eye Trackers are supported: **Tobii X60, Tobii X120, Tobii T60, Tobii T120, Tobii T60 XL, Tobii TX300, Tobii X1 Light, Tobii X2-30 and Tobii X2-60.** Models that are not supported are: **Tobii 1750, Tobii X50, Tobii 2150, Tobii 1740 (D10), Tobii P10, C-Eye, PCEye, Tobii IS-1 and Tobii Rex.**

To run an experiment with a Tobii, you will need the `TobiiController` module and the `TobiiSprite` module. Optionally you can also include the `TobiiWatcher` module.

When developing your experiment without an eye tracker hooked up, you can use the `TobiiSimController`.

To include these modules, use the format:

```
import kelpy.tobii.nameOfModule
```

Check out the tobii examples under the folder `examples/demo-tobii` for complete experiment setups.

1.4.1 TobiiController

The tobii controller is the interface between kelpy and your Tobii. It controls connecting to your Tobii as well as gathering eye data that is then written to an output file. Here is the basic setup for initializing a Tobii:

```
# this creates a TobiiController that calls the Tobii SDK code
tobii_controller = TobiiController(screen)

# this searches for the tobii eyetracker that is connected.
# It times out based on the given amount of seconds (the default is 1,000 seconds) and exits this pro
tobii_controller.wait_for_find_eyetracker(3)

#set the name of the data file that will output all of the Tobii data
tobii_controller.set_data_file('testdata.tsv')

#activate the first tobii eyetracker that was found
tobii_controller.activate(tobii_controller.eyetrackers.keys()[0])
```

For each trial, you will then need to start and stop the eye tracker to prevent data collection between trials:

```
#start eye tracking; this can be called later in your script once all the experiment setup is complet
tobii_controller.start_tracking()

trial_time = 5.0
for event in kelpy_standard_event_loop(screen, Q, dos, throw_null_events=True):

        #end the trial once the time is up
        if (time() - start_time > trial_time):
                break

        #this is set specifically for the tobii controller,
        #otherwise the program hangs since the text file is not closed
        if event.type == QUIT:
                 tobii_controller.close_data_file()
                 tobii_controller.destroy()
```

```
        #stop eye tracking when you no longer need the eye data for the trial
        tobii_controller.stop_tracking()
```

As shown in the sample code above, you may also need to check for any attempts to exit out of the experiment before it is complete; if there are any, make sure you close the data file (using `tobii_controller.close_data_file()`) and disconnect from the Tobii (using `tobii_controller.destroy()`) before ending the program. These two function calls should also be included at the end of your script.

1.4.2 TobiiSprite

A `TobiiSprite` is a variation of a `CommandableImageSprite`. You can pass all the same arguments to a `TobiiSprite`; the only difference is that you must also pass the TobiiController object as well:

```
sprite = TobiiSprite(screen, init_position, imagepath, tobii_controller, rotation=0, scale=1.0, brigh
```

1.5 Credits

Steven T. Piantadosi, Matthew McGovern, and Amanda Yung are the primary contributors to this project.