psychbench

Element type programming manual

Quick element types are custom code for a limited number of experiments where you don't need much flexibility or much of an interface for other users. Read sec 1 (≈ 15 pages) if you want to make a quick type.

Durable element types (optional) are additions to your library for an open range of users and experiments where the code needs to be more flexible (object properties) and have more of a user interface (error checking, documentation, etc.). Add reading sec 2 if you want to make a durable type. Skip otherwise.

Core property reference: Sec 3 is reference for core properties that can be useful in element type code. Read only as needed.

Contents

1. Quick element types 3

1.1. What you need 3

1.2. Files and folders 4

1.3. Type-Specific / Core 4

1.4. Making and managing an element type 5

Making an element type – *newPbType* 5

Copying/Forking an element type 5

Editing an element type 5

Renaming an element type – *renamePbType* 5

Removing an element type 6

Sending/Receiving an element type 6

1.5. Element type code 6

Type scripts 6

Objects/Properties and Variables in type scripts 7

Functions/Commands in type scripts 9

Test using debug mode! 9

1.6. Setup – *open* script 10

1.7. Running – *runFrame* script 10
Object start/end 11
Time 11
1.8. Wrapup – *close* script 13
Experiment results output 13
1.9. Cleanup on error – *catch* script 13
1.10. Showing a display 14
Recap: Showing a display without PsychBench 14
Core display functionality and rules 14
Distance units – *element_deg2px*, *element_px2deg* 17
Drawing a display to the experiment window 17
Direct draw method 18
(Texture method) 18

2. Durable element types (optional) 20
2.1. Input properties 20
Default values – *typeOptions* variable *inputPropertyDefs* 20
String properties – *var2char*, *var2string* 21
Error checking input properties 21
2.2. Time/Memory in *open*: *open1*, *2*, *3* type scripts and tools 21
open1, *2*, *3* scripts 21
Conserve time – *element_doShared* 22
Conserve memory in *open2* – *element_setShared* 23
2.3. Showing a display – Texture method 23
Making textures 23
Drawing textures to the experiment window 24
2.4. Object sleep/wake 27
2.5. Staircased properties 27
2.6. Adjustable properties 27
2.7. Documentation 31
2.8. Name conventions 32

3. Core property reference 33
3.1. Core properties – Visual element objects (*this.<property>*) 33
3.2. Core properties – Trial objects (*trial.<property>*) 36
3.3. Core properties – Experiment object (*experiment.<property>*) 37
3.4. Core properties – Screen objects (*devices.screen.<property>*) 38

1. Quick element types

Quick element types are custom code for a limited number of experiments where you don't need much flexibility or much of an interface for other users. Read this section (≈ 15 pages) if you want to make a quick type.

If the element type library doesn't have the stimuli you need for an experiment, first make sure PsychBench is up to date using *pb_update*. You can also email us at contact@psychbench.org to request an element type.

Otherwise you can write your own code for stimuli in MATLAB + Psychtoolbox. Almost never do you just need code that runs once in some trial. Generally you want code that repeats across multiple trials and with parameters you can control (e.g. stimulus features). You also often need part of your code to do work before or after trials so the part during trials can run fast and not impact frame rate. And you don't want to re-invent the wheel when it comes to all the core functionality PsychBench already handles (timing, visual options, staircasing, results output, etc.).

To make all this easy, the approach is to write your code as an element type in your library. You can then use it anywhere by making objects of the type, parameterize and get output from it using object properties, and have it click with all core functionality. All the element types that come with PsychBench are open source, so you can use them as examples and/or build off them.

To keep this manual simple, we focus on making element types that show visual stimuli. However, the element type programming framework allows for many other types too: auditory, response handler, adjuster, etc. For more information please email contact@psychbench.org.

1.1. What you need

You should know the basics of showing a display in Psychtoolbox code without PsychBench: opening an on-screen window using *Screen('OpenWindow')*, drawing to its buffer using *Screen* draw functions, flipping the buffer using *Screen('Flip')*, and repeating the process each frame when the display should change. You won't use all these commands in PsychBench element type code but the concepts are important. Psychtoolbox tutorials are available, for example at https://peterscarfe.com/ptbtutorials.html. Aside from that, the amount of MATLAB programming knowledge you need just depends on what you want to do.

Object-oriented framework

PsychBench uses its own framework for objects, which are just represented by structs.
You don't need to know anything about MATLAB's object-oriented framework.

1.2. Files and folders

- **Element types that come with PsychBench – */element types***
  Code for element types that come with PsychBench is in *<PsychBench folder>/element types*. Each element type is in a subfolder with name = type name. Documentation for these types is online at www.psychbench.org/docs. All these element types are open source, so you can use them as examples and/or build off them. Don't edit them directly since your edits would be lost the next time you update (sec 1.4).
- **Element types you make – Local element types folder**
  For custom element types you make or receive, both code and documentation go in your ***local element types folder***. Again each element type is in a subfolder with name = type name. You set the location for your local element types folder when you install PsychBench. You can move it anytime later—if PsychBench can't find it, it will just ask for a new location. It's always outside the main PsychBench folder, so updating or uninstalling PsychBench doesn't affect your code.
- **Tools – */element type programming***
  All tools for making element types are in *<PsychBench folder>/element type programming*. The root folder contains tools for making/managing element types. Subfolders contain tools for use in element type code. An overview of files is available through the *pb* help command. All functions have help text by typing *help <function>* at the MATLAB command line.

1.3. Type-Specific / Core

The basic principle in making an element type is the separation between type-specific and core:

Type-specific: Type-specific properties are properties only objects of the type have.
And type-specific functionality is everything new which the element type adds to PsychBench, typically based on its type-specific properties. If you go to any element type page on the PsychBench website, everything type-specific about it is documented there (e.g. *gabor*).

Core: Core properties are properties different element types have in common, or properties of non-element objects like *trial*, *experiment*, etc. And core functionality is everything that is common across element types or handled in non-element objects, typically based on core properties. Examples are starting/ending elements based on core properties *start*/*end*, experiment results

output based on core properties *report/info*, etc. Everything core is documented on pages like All elements, All visual elements, etc. on the PsychBench website.

- When you make an element type, you only need to make its type-specific properties and functionality. PsychBench adds and handles all core properties/functionality automatically, so you don't need to.
- In some cases element type code needs to call PsychBench functions to tell PsychBench when to apply core functionality, or to interact with it. See specific sections below.

1.4. Making and managing an element type

Making an element type – *newPbType*

To make a new element type, start by calling *newPbType* at the MATLAB command line and input a name. This makes a subfolder in your local element types folder (sec 1.2), and sets up a *typeOptions* script as well as a user documentation text file there. You can then write code for the element type and save it in the folder. **For a quick type you can generally ignore the *typeOptions* script and user documentation.** For a durable type you need to work with them (secs 2.1, 2.7).

Copying/Forking an element type

Optionally you can make a new element type using an existing type as a starting point. To do this, input the name of the source type in a second input to *newPbType*. *newPbType* sets up the new type as a copy of the source type. You can then edit the copy independent of the source. You can build off types that come with PsychBench and local types in this way.

Editing an element type

To edit an element type you have made, just edit its files. PsychBench automatically sees all additions and edits.

- Don't edit the element types that come with PsychBench in *<PsychBench folder>/element types* since your edits would be lost the next time you update. Instead use *newPbType* to make a copy in your local element types folder and edit that.

Renaming an element type – *renamePbType*

If you want to rename an element type you have made, generally all instances of the type name in its files and folders need to be changed. You can use the tool *renamePbType* to automate this.

Removing an element type

To remove an element type you have made, just delete its folder or move it out of your local element types folder.

Sending/Receiving an element type

To share an element type with other people, just zip its folder and email it. Also include any other files the type needs. To add an element type you have received, just copy its folder into your local element types folder, then call *pb_addPath* to refresh PsychBench.

If you would like to contribute an element type to the library that comes with PsychBench, please email us at contact@psychbench.org. We would be happy to credit you. And thank-you!

1.5. Element type code

Once you have used *newPbType* to set up a subfolder in your local element types folder, it's time to write the code for the element type and save it there. This is the code PsychBench will call during any experiment to run any object of the type. It runs independently for each object of the type in an experiment. Generally each object can have different property values and/or run in different contexts (e.g. at different positions on screen), which means the code needs to be flexible. To do this, it reads property values of the object it's running for, and maybe information from other core objects (e.g. *trial*, *experiment*, *screen*) and functions (e.g. Psychtoolbox *GetSecs*), and uses it as instructions or parameters. (For a quick type which will only be used in a limited number of experiments, you might hard-code more and have less flexibility.)

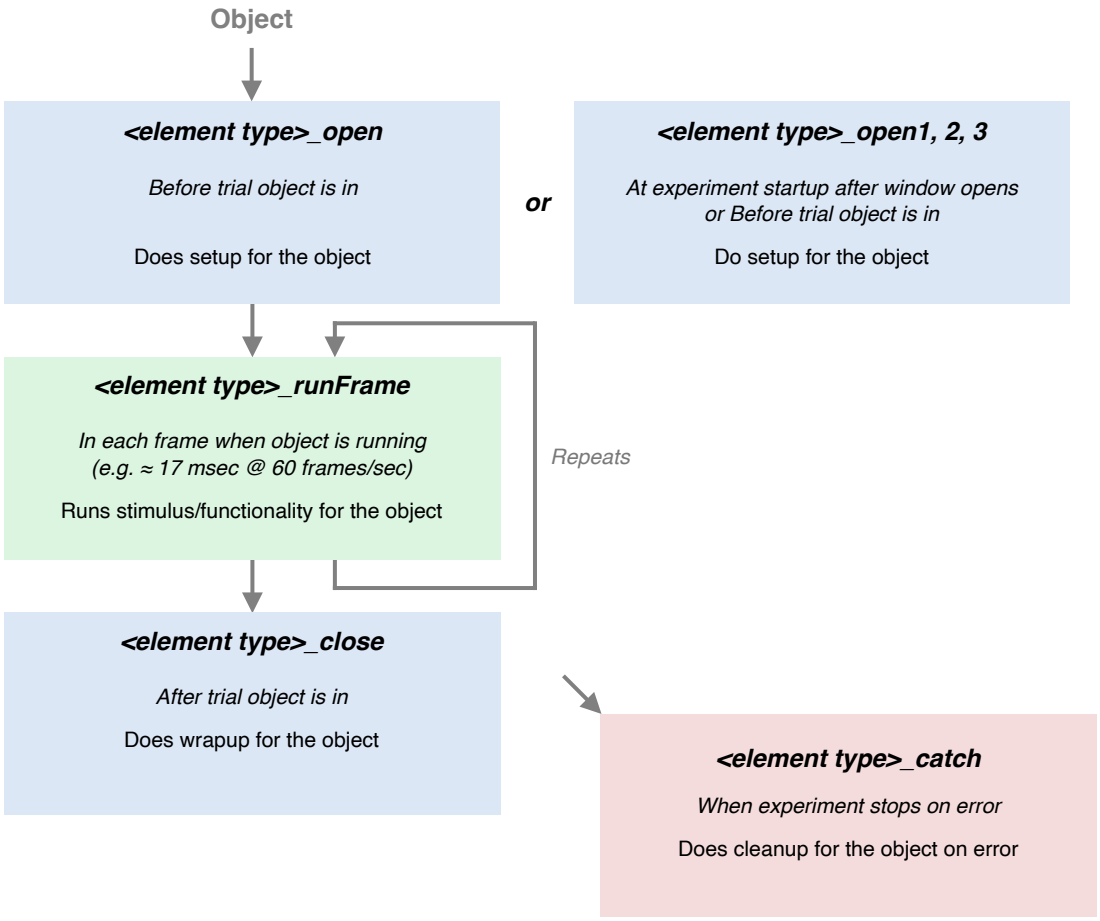
This section looks at the basic framework for writing element type code. If examples would be helpful, you can look ahead to Examples 1, 2.

Type scripts

Each element object in an experiment runs in one trial. However, an element type often needs different code to run at different points in the experiment—for example, code to do any slow preparation of an object before trials, faster code to run it during its trial when everything has to happen in real time, etc. So code is structured into different scripts called ***(element type scripts)*** , each of which runs at a different point. Type scripts have specific names with the form *<element type>_<when to run>*, e.g. for the *picture* type: *picture_open*, *picture_runFrame*, *picture_close*, etc. Overall type scripts form a workflow that each object passes through during an experiment. **Each type script is optional—if an element type has no code for a given point in the experiment, you can omit that script.**

Following is the big picture. Later sections look at each script specifically.

Figure. Type scripts



Objects/Properties and Variables in type scripts

Element object a type script is running for – *this*

Element type code runs once for each object of the type in an experiment. Each time a type script runs, it is in a workspace that PsychBench sets up with objects that the script can read from and write to. The main one is the object of the type the script is currently running for, which is present in a variable called *this*. To read and write properties of *this*, use standard dot syntax *this.<property name>* .

Important in making an element type is the distinction between type-specific and core properties (sec 1.3):

- ***Type-specific properties*** are properties only objects of the type have. The element type handles these.
- ***Core properties*** are properties common across objects of different types. PsychBench handles these, so the element type doesn't need to.

Recall there is also the distinction between input/record properties:

- ***Input properties*** can be set by users when building experiments.
- ***Record properties*** cannot be set by users. Instead they are set by PsychBench or the element type during experiments. Users can ask to see them in experiment results output.

So there can be type-specific and core input properties, and type-specific and core record properties. Here is how this all works when making an element type:

▷ Type scripts can read and write type-specific properties of *this*.

For a quick type, define type-specific properties just by setting them: The user must set all type-specific input properties when building the experiment. Type scripts set type-specific record properties at any point. (For a durable type, you pre-define type-specific input properties in the *typeOptions* script to give them default values, sec 2.1.)

▷ Type scripts can only read (not write) core properties.

PsychBench automatically adds all core properties to objects of the type you're making. The user can set core input properties or leave them at default —either way they are all present in type scripts. PsychBench sets all core record properties during the experiment.

▷ Type scripts can make local variables but they are not retained when the script ends.

Taken together, type scripts do three things with the object *this*:

- **Read input properties:** Type scripts read input properties of *this* and uses them as instructions or parameters. Usually type-specific input properties, but they can also read core input properties if needed.
- **Write to type-specific properties to carry information between type scripts:** Local variables type scripts make are not retained. So the only way to retain information between type scripts in the workflow is to write to type-specific properties of *this*. Usually record properties, but type scripts can also change input property values if convenient.
- **Write to type-specific record properties for experiment results output:** If type scripts generate or obtain information that the user might want to see, write it to type-specific record properties of *this*. Users can see those properties in experiment results output using core input property *this.report* (sec 1.8).

Other objects – *trial*, *experiment*, *devices.screen*, ...

Type scripts can also read properties of non-element objects that are present in the workspace. These are core objects, so type scripts cannot write to their properties:

- *trial* contains the trial object for the trial the element is in.
- *experiment* is the experiment object for the experiment.
- *devices* is a struct with fields containing device objects the element object uses. Field names = device type names. The relevant one for a visual element type is *devices.screen*.

Functions/Commands in type scripts

Generally you can call MATLAB functions, Psychtoolbox functions, and PsychBench tools in type scripts. The tools in *<PsychBench folder>/element type programming* subfolders are ***type script tools*** which are specifically for use in type scripts. Note the following MATLAB tools:

- You can throw errors for users at any point using the usual MATLAB *error*. The most common example is errors from checking input property values in the *open* script (sec 2.1). The experiment will stop and display the error message at the MATLAB command line.
- You can throw warnings using MATLAB *warning*. The experiment won't stop but the user will see them at the MATLAB command line after the experiment window closes.
- The *return* command works to exit a type script before its end.
- Type scripts can have local functions. You can also use persistent variables in type script local functions. Note PsychBench automatically clears them when the experiment ends or stops on error. For a durable type see also *element_doShared* (sec 2.2).

Test using debug mode!

By default some error information for element type code is disabled. So to test properly, you must turn debugging on by typing *debugPbTypes* at the MATLAB command line. You can turn it off by typing *debugPbTypes* again. Debugging turns off automatically if you type *clear all* or end the MATLAB session.

▷ Before testing element type code, turn debugging on by typing *debugPbTypes* at the MATLAB command line. Don't use debugging during real experiments since it can reduce frame rate.

1.6. Setup – *open* script

The *<element type>_open* script runs once for each object of the type to do any type-specific setup needed (e.g. loading data from a file, opening textures for a display, etc.). It runs in the inter-trial interval before the trial the object is in. Object input properties start out as set by the user. The purpose of *open* is to keep slow setup work out of the trial where it could cause dropped frames (*runFrame* below).

open should at least be fast enough to run in an inter-trial interval (e.g. << 0.5 sec). If you have setup work that is slower than that, you can optionally split *open* into three scripts:

open1 – runs at experiment startup
open2 – runs at experiment startup *or* before the trial, depending on user settings
open3 – runs before the trial

However, If so you need to consider time and memory usage. See sec 2.2 for more information, or if you want to know more about efficiency in *open* scripts generally.

1.7. Running – *runFrame* script

Frames

Time during trials is divided into short increments called ***frames***. Frames are locked to refresh of the screen the experiment is running on: frame transition is always at screen refresh, and nominal frame rate = screen refresh rate. If a frame is too short for the processing that needs to run in it, it extends to multiple refresh intervals—***dropped frames***. A common screen refresh / frame rate is 60 frames/sec (interval = 1/60 ≈ 17 msec).

Frames are the time basis for most events during trials. e.g. PsychBench checks to start/end objects once per frame, dynamic displays change on screen up to once per frame at screen refresh, etc. So, if something causes a frame to drop (like a *runFrame* script taking too long), it delays everything else too.

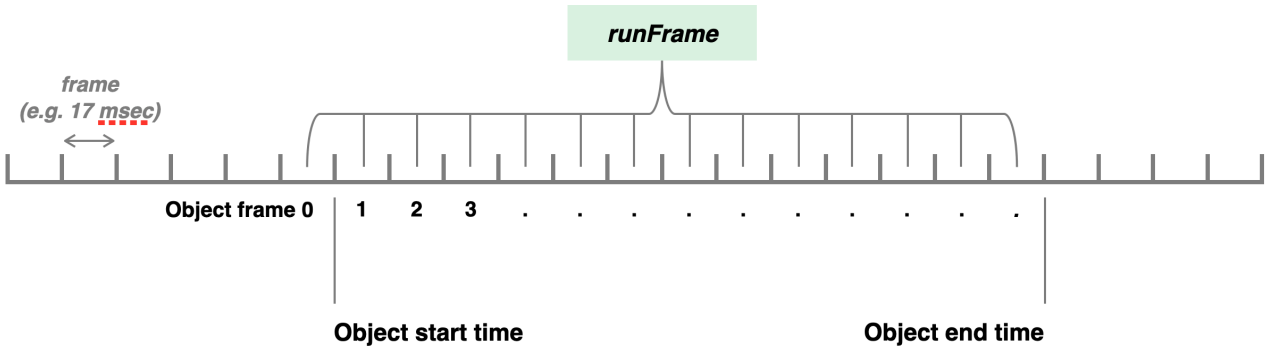
See www.psychbench.org/docs/timingprecision for more information on frames and dropped frames.

The *<element type>_runFrame* script **repeats** once each frame (e.g. at 60 frames/sec) during trials for each object of the type that it is running. Property values for an object are passed from one iteration of *runFrame* to the next. The code in *runFrame* implements the stimulus/functionality of the object (e.g. drawing an image to the experiment window to show on screen next frame).

runFrame can do things for the current frame as well as cue things for the next frame. The first iteration of *runFrame* runs in the frame *before* the object starts (object "frame 0"), with object start time *this.startTime* marked at that frame end / next frame start. This allows the script to cue things for object frame 1 if needed. The last iteration of *runFrame* simply runs in the last object frame, with object end time *this.endTime* marked at that frame end. If you need to run different code or exclude code in frame 0 and/or last frame, you can check core properties *this.isStarting* and *.isEnding*, which = `true` when the current iteration of *runFrame* is in frame 0 and last frame respectively, else = `false`.

▷ Each iteration of *runFrame* should run within a frame (e.g. ≈ 17 msec at 60 frames/sec), so it needs to be fast else dropped frames will occur. So, do as much setup as possible in *open* (sec 1.6) and wrapup in *close* (sec 1.8) to minimize work in *runFrame*.

Figure. runFrame *script during frames*



Object start/end

Users set when an element object starts and ends using core input properties *this.start/end*, and PsychBench starts and stops calling the *runFrame* script for the object based on this. Optionally the *runFrame* script can cue the object to end earlier by calling *element_end*. From the perspective of the user the object ends "on its own". The object will then end at frame end and the current iteration of *runFrame* will be the last. Generally use *element_end* when the object has nothing left to do, e.g. it is showing a movie file and the file ends.

Time

The *runFrame* script often needs to work with time values, e.g. to generate the next image in a dynamic display based on current time.

System time

Times that users see in experiment results output are generally relative to trial 1 start. However, times in element type code are generally in ***system time*** (sec) like returned by Psychtoolbox *GetSecs*. This makes them easy to use with Psychtoolbox functions.

Absolute system times are not meaningful. However, relative system times are meaningful—for example:

> object start time (system) – trial start time (system) = object start time (relative to trial start)

▷ In element type code all core record properties containing times are in system time (e.g. *this.startTime/endTime*, etc.).

Current time

There are different measures of "current time" you can use in *runFrame*. The difference between them is small, but can matter if precise timing is important:

- ***GetSecs* (instantaneous time)**
  Psychtoolbox function *GetSecs* returns the time when it's called (system time, sec). Use *GetSecs* if you need to know current time *within* the current frame.
- ***trial.nextFrameTime***
  Trial object property *nextFrameTime* contains expected mid time of *next* frame (system time, sec). This is slightly more accurate when cueing functionality that will occur next frame, including generating the next image in a dynamic display.

Object start and end times

Start and end time of the object a script is running for is available in core properties *this.startTime* and *.endTime* (system time, sec). By default object start time is measured as object frame 1 start (after the first iteration of *runFrame* in frame 0), and end time as last object frame end (after the last iteration of *runFrame*), and the times are available from those frames through to the *close* script. A common example of using object start time is if you need current time relative to object start, e.g. `trial.nextFrameTime–this.startTime`. See sec 1.10 Example 2.

* During *runFrame* these properties contain *expected* frame transition times. PsychBench replaces them with actual measured times before the *close* script. So if your code needs to use these properties for purposes related to experiment results output, wait until *close*. See reference documentation at the link above for more information if needed.

Other times

Other times are available in core properties which you can access if needed, e.g. other measures related to current time in *trial.nextnextFrameTime*, *trial.frameStartTimes* and *.frameTimes*, trial start and end time in *trial.startTime* and *.endTime*, and experiment start time in *experiment.startTIme*. Nominal frame interval is also available in *experiment.frameInterval* (e.g. ≈ 17 msec at 60 frames/sec).

1.8. Wrapup – *close* script

The *<element type>_close* script runs once for each object of the type to do any type-specific wrapup needed (e.g. closing textures). It runs in the inter-trial interval after the trial the object is in. Note *close* runs even if the object didn't run (didn't start) in the trial. If *close* code needs to know whether the object ran, it can check core property *this.ran* (= `true` if ran).

Like *open* (sec 1.6), the purpose of *close* is to keep slow work out of the trial (*runFrame* above) where it could cause dropped frames. *close* should at least be fast enough to run in an inter-trial interval (e.g. << 0.5 sec). Most commonly use Psychtoolbox *Screen('Close')* to close textures you opened (sec 1.10).

Note in cases where memory usage would increase open-endedly as an object runs, some or all wrapup code can go in *runFrame* instead. For example, if *runFrame* repeatedly opens new textures, it should close them when it is done with them as it runs (see for example element type *movie*).

Experiment results output

Users tell PsychBench what they want to see for an object in experiment results output using core input properties *this.report* and *.info*. This is core functionality which PsychBench handles automatically. The only thing to know for quick types is that PsychBench takes any property values for results after element type code is done. So you have all the way to the end of *close* to write record property values users might want to see.

1.9. Cleanup on error – *catch* script

The *<element type>_catch* script is like *close* except it runs once for each object of the type if the experiment stops on error to do any cleanup needed. It **replaces** *close* in that case. *catch* doesn't run for an object if *close* has already run for it. *catch* also doesn't run for an object if the error occurred before any type scripts ran for it (if no type-specific code ran, no type-specific cleanup is needed).

For convenience PsychBench does a lot of cleanup on error automatically, so often you don't need a *catch* script at all. In particular PsychBench automatically closes all textures on error, so you don't need to write *catch* code for that (sec 1.10).

If the error occurred during the element type's code, any changes made to *this* up to that point are retained for *catch*. In addition to the usual objects like *this*, the MATLAB *MException* object representing the error is also present for *catch* in a variable *X*.

1.10. Showing a display

Recap: Showing a display without PsychBench

The basic approach to showing a display in Psychtoolbox code without PsychBench is:
(1) Use *Screen('OpenWindow')* to open an on-screen window; (2) Use Psychtoolbox draw functions to draw a display to the window buffer (*Screen('DrawDots')*, *Screen('DrawTexture')*, etc.); (3) Call *Screen('Flip')* to move what you drew from the buffer to the screen at next screen refresh. Repeat (2)–(3) each time the display should change, up to once per screen refresh.

Core display functionality and rules

The main difference between Psychtoolbox code and PsychBench element type code is that PsychBench handles a lot of things automatically as core functionality so your code doesn't need to. However, there are some rules your code needs to follow in order to play well with it. See later for examples.

Experiment window

PsychBench opens the Psychtoolbox imaging pipeline and on-screen window (***experiment window***) at experiment startup. It also closes them at experiment wrapup or if the experiment stops on error. Every display shows in the experiment window. Experiment window number (pointer/handle) is available in core property *this.n_window* (also *devices.screen.n_window*) if needed. Window size and center (px) are available in *devices.screen.windowSize, .windowCenter* if needed.

▷ Don't call Psychtoolbox functions like *Screen('OpenWindow')* or *Screen('CloseAll')* in element type code. This is core functionality which PsychBench handles.

Window options

PsychBench sets general options for the experiment window at startup, e.g. alpha blending (see below).

▷ Generally don't change options for the experiment window in element type code since that would also affect other objects. If you do, change the options back before the type script ends.

px units

Object properties users set relating to distance on screen typically use degrees visual angle or other distance units (deg, deg-, cm, ww, wh, wwh, px). However, core properties as well as PsychBench tools and Psychtoolbox functions in type scripts always use px units.

▷ Use px units in element type code. You can use *element_deg2px* in the *open* script to convert all type-specific property values in deg or other distance units to px (see below).

RGB units

PsychBench sets all Psychtoolbox functions to use RGB values in the range 0–1: 0 = no intensity, 1 = full intensity. This is also the convention for users when making experiments in PsychBench.

▷ Use RGB values between 0–1 (not 0–255) in element type code.

Transparency

PsychBench enables alpha blending for the experiment window using standard linear blend factors GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA (and blend equation GL_FUNC_ADD).

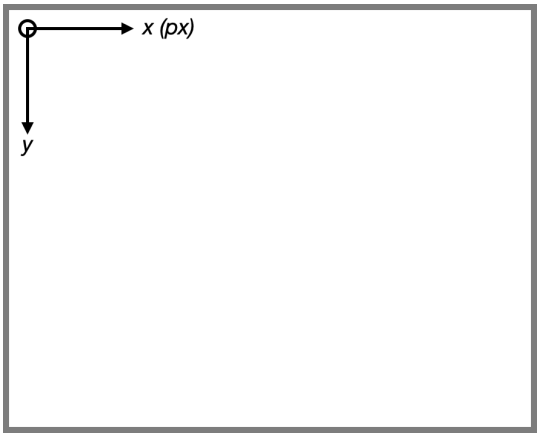
▷ Any transparency (RGBA A < 1) in an object display will work.

this.position

When users set core input property *this.position*, `[0 0]` = experiment window center and default units are deg. However, PsychBench tools and Psychtoolbox functions always use `[0 0]` = top left and units px.

▷ Before the *open* script, PsychBench translates *this.position* to `[0 0]` = experiment window top left (px) to facilitate use with Psychtoolbox functions.

Figure. Display coordinate system in element type code



Core display options for the object

There are a number of core input properties users can set for display options, e.g. *this.position*, *.nn_eyes*, *.rotation*, etc.

▷ You must use the texture method for showing a display if you want PsychBench to apply core display options automatically. However, if you use the direct draw method, your element type code can still read and apply any of them manually. See below.

Experiment window buffer flip

The *runFrame* script runs once per frame for each running object. Frames are locked to screen refresh: frame transition is always at screen refresh, and nominal frame rate = screen refresh rate (sec 1.7). PsychBench automatically flips the experiment window buffer at the end of each frame in case any visual object has changed its display.

▷ Don't call Psychtoolbox *Screen('Flip')*, *Screen('DrawingFinished')*, or other flip-related functions in element type code. This is core functionality which PsychBench handles.

▷ To maintain an object display you must draw an image to the experiment window every frame, even for a static display where the image doesn't change.

Close textures

▷ Use Psychtoolbox *Screen('Close')* to close textures you open when done with them. However, PsychBench automatically closes all textures if the experiment stops on error, so you don't need to write *catch* script code for that.

Distance units – *element_deg2px*, *element_px2deg*

Core properties as well as PsychBench tools and Psychtoolbox functions in type scripts always use px units. However, you'll often want to make object input properties settable using degrees visual angle or other distance units (deg, deg-, cm, ww, wh, wwh, px). To do this, use *element_deg2px* at the top of the *open* script: Call *element_deg2px* once for each type-specific property value that could use deg. It converts all numbers in the property value to px based on screen height and distance for the experiment. Any other data type it returns unchanged (doesn't throw an error, so you can use it before error checking property values, sec 2.1). Then write the px value back to the property for use in the rest of your code. *element_deg2px* also handles property values in `{value, "unit"}` form specifying other units, and compound units with any exponent on the distance unit (e.g. cycles/deg, exponent = –1). Type *help element_deg2px* for usage.

Also on the output side: For record properties that users might want to see in experiment results output, you can convert them from px to deg or other units in the *close* script. *element_px2deg* does this, with similar usage.

▷ PsychBench converts core properties like *this.position* to px automatically, so you don't need to for them.

Drawing a display to the experiment window

Here's the general approach: In the *runFrame* script draw an object display to the experiment window each frame (each iteration of *runFrame*) to start or maintain it. Typically this just means put the general draw code in *runFrame* to repeat it each frame. Draw different images in some or all frames for a ***dynamic display*** (e.g. movie), or redraw the same image for a ***static display*** (e.g. picture). PsychBench automatically flips the window buffer to show what was drawn at next frame start (screen refresh). To stop showing a display before the object ends, just stop drawing (call *element_end* if the object has nothing more to do).

By default object start time *this.startTime* is measured as object frame 1 start (after the first iteration of *runFrame* in frame 0), and end time *this.endTime* as last object frame end (after the last iteration of *runFrame*). For precise timing you should align display start/end with these times. That means draw the first image in object frame 0 (first iteration of *runFrame*) and omit drawing in last frame (last iteration). The display will then start at frame 1 start and end at last frame end. You can use a `~this.isEnding` check in *runFrame* to do this (see examples below). If you don't, you will just get one frame of display showing after the object end time (≈ 17 msec at 60 frames/sec).

See sec 1.7 on *runFrame*.

There are two methods to draw a display to the experiment window: the ***direct draw method*** and the ***texture method***...

Direct draw method

In the direct draw method, just draw the display to the window using Psychtoolbox functions. Experiment window number (pointer/handle) is available for this in core property *this.n_window*. Direct draw is simple, so many quick types use it. The disadvantage of direct draw is that it doesn't let PsychBench apply any core display options for the object (*this.position*, *.rotation*, *etc.*). Your code needs to apply the ones you want manually using Psychtoolbox functions. Usually you will want to apply at least *this.position* by drawing the display centered at that point.

Direct draw can also be used for any element type (including a durable type) where you want to override any of the core display options or need full control of drawing to the experiment window.

(Texture method)

In the texture method, make the display in the form of one or more textures and give them to PsychBench to draw. This is a little more work (not much). However, a big advantage is PsychBench applies all core display options for the object automatically. The texture method is standard for durable types, but it can also be useful for quick types that need core display options beyond just *this.position*. For information on the texture method, see sec 2.3.

Example 1. Direct draw method – Static display

The object shows a rectangle. The user can set input properties *dims* `[w h]` (deg) and *color* (RGB).

open: The *open* script runs in the inter-trial interval before the trial the object is in. Here it converts type-specific input property *dims* from deg or other distance units to px. It then calculates the rect `[tl tr bl br]` on screen for the rectangle based on *dims* and core input property *position*, and saves it in record property *rect* for use later in *runFrame*. This is all in coordinates with `[0 0]` = experiment window top left and units px. We do this in *open* to keep work out of frames.

```
this.dims = element_deg2px(this.dims);

dims = this.dims;
position = this.position;

rect = [0 0 dims]+repmat(-(dims+1)/2+position, 1, 2);

this.rect = rect;
```

runFrame: *runFrame* repeats once each frame when the object is running. Here it calls Psychtoolbox *Screen('FillRect')* to draw the rectangle to the experiment window to show on screen next frame. For precise timing we check core property *this.isEnding* to omit drawing in the last object frame so the display doesn't show in the frame after the object ends.

```
if ~this.isEnding
    rect = this.rect;
    color = this.color;
    n_window = this.n_window;

    Screen('FillRect', n_window, color, rect)
end
```

Example 2. Direct draw method – Dynamic display

The object shows a dot moving in a circle. The user can set parameters in input properties *radius* (deg), *velocity* (deg/sec), *dotSize* (deg), and *color* (RGB).

open: The *open* script runs in the inter-trial interval before the trial the object is in. Here it converts type-specific input properties from deg (deg/sec for *velocity*) or other distance units to px.

```
this.radius = element_deg2px(this.radius);
this.velocity = element_deg2px(this.velocity);
this.dotSize = element_deg2px(this.dotSize);
```

runFrame: *runFrame* repeats once each frame when the object is running. Here we calculate dot position for next frame using expected time of next frame relative to object start. The display is centered at object position. We then use Psychtoolbox *Screen('DrawDots')* to draw the dot to the experiment window to show on screen next frame (note *DrawDots* needs position as a column vector). For precise timing we check core property *this.isEnding* to omit drawing in the last object frame so the display doesn't show in the frame after the object ends.

```
if ~this.isEnding
    r = this.radius;
    v = this.velocity;
    dotSize = this.dotSize;
    color = this.color;
    startTime = this.startTime;
    position = this.position;
    n_window = this.n_window;
    nextFrameTime = trial.nextFrameTime;

    t = nextFrameTime-startTime;
    a = v/r*t;
    p = [r*cos(a) r*sin(a)]+position;
    p = transpose(p);
    Screen('DrawDots', n_window, p, dotSize, color)
end
```

2. Durable element types (optional)

Durable element types are additions to your library for an open range of users and experiments where the code needs to be more flexible (object properties) and have more of a user interface (error checking, documentation, etc.). Read this section (2) in addition to sec 1 if you want to make a durable type. Skip otherwise.

2.1. Input properties

Default values – *typeOptions* variable *inputPropertyDefs*

Input properties of objects are properties users can set when making experiments. Your code reads them as instructions or parameters (sec 1.5). For a quick type the user just sets them all. However, for a durable type pre-define all type-specific input properties using *typeOptions* script variable *inputPropertyDefs*. This is a cell array with each row listing a property name and default value. This does three things:

- Users can omit setting type-specific input properties to leave them at default, just like core input properties. Specifically, If the user doesn't set a property or leaves it = [], PsychBench sets it to its default value before the *open* script.
- PsychBench throws an error if the user sets an input property that is not defined.
- For experiment results output based on core input property *this.report* set by the user, PsychBench takes values for defined input properties before the *open* script runs. This means your code can change input property values if convenient and in results the user will still see the value they set. (For record properties in results, PsychBench always takes them after *close*, sec 1.8.)

▷ [] means default for an input property defined in *typeOptions* variable *inputPropertyDefs*, so if [] is a possible value then it must also be the property's default value.

▷ PsychBench adds core properties automatically, so you don't need to define them in *typeOptions* variable *inputPropertyDefs*.

String properties – *var2char*, *var2string*

For a durable type it's a good idea to allow text input properties to accept both `'x'` (char / cell array of char) and `"x"` string data types. However, in element type code you may want to use `'x'` since many MATLAB functions only output this data type and Psychtoolbox functions only use it.

The tool *var2char* automates doing this. At the top of the *open* script, give each input property that could contain a text value to *var2char*. If it's `"x"`, *var2char* converts it to `'x'`. If it's anything else, *var2char* leaves it unchanged. This also works on each cell in a cell array (see function help). Then write the standardized value back to the property for use in the rest of your code. Note *var2char* accepts any data type and size, so you can use it before error checking input property values (below).

Also on the output side: For record properties that users might want to see in experiment results output, it's nice to convert them to `"x"` strings in the *close* script. The tool *var2string* does this, with similar usage.

▷ PsychBench converts core properties containing text to/from `'x'` automatically, so you don't need to for them.

Error checking input properties

For a durable type an important part of the user interface is checking that each input property value is okay and throwing a helpful error message if not. Generally do this near the top of the *open* script (you can run input properties through *element_deg2px* and *var2char* first). Lots of little tools to help with error checking are in *<PsychBench folder>/element type programming/general*. See the *open* script for any element type that comes with PsychBench for an example.

▷ PsychBench error checks core properties automatically, so you don't need to for them.

2.2. Time/Memory in *open*: *open1*, *2*, *3* type scripts and tools

open1, *2*, *3* scripts

The regular *open* script runs in the inter-trial interval before the trial the object is in. Since *close* runs after the trial, the object only uses memory (RAM or VRAM) during its trial. This is the default in case the objects in an experiment would cause an out-of-memory error if all opened at the same time at experiment startup, depending on object types used and system.

You can optionally split *open* into any or all of three scripts *open1*, *open2*, *open3* which run at different points ranging from experiment startup to before trial. This may be necessary if some *open* code is too slow to run in an inter-trial interval (e.g. << 0.5 sec). *open1*, *2*, *3* run in that order and property values of *this* carry between them as usual. How you divide your code between them

depends on its time and memory needs. *open1* is for code that is not memory-intensive, *open3* is for code that is fast, and *open2* is for code that requires a tradeoff between time/memory:

- ***open1*** – Runs at experiment startup after the experiment window opens[1]
  Time: Code can be slow or fast
  Memory: Code cannot be memory-intensive
- ***open2*** – Runs at experiment startup *or* before the trial, depending on how the user sets core input property *preload* (`true` = experiment startup (default), `false` = before trial). i.e. it either works like *open1* or *open3*, though it always runs after any *open1* and before any *open3*. *open2* is for code that requires a trade-off between time and memory. This allows the user to customize the tradeoff based on their experiment and system.
  Time: Code is slow *and*
  Memory: Code is memory-intensive (RAM or VRAM)
- ***open3*** – Runs in the inter-trial interval before the trial the object is in
  Time: Code cannot be slow (> 0.5 sec)
  Memory: Code can be memory-intensive or not

All the durable element types that come with PsychBench use *open1*, *2*, *3*. A common approach is to put basic formatting and error checking properties in *open1*, slowish code that uses RAM in *open2* (e.g. generating image matrixes needed for textures), and making textures in *open3* (since making textures is relatively fast if you already have the image data saved in a property from *open2* or are just using Psychtoolbox draw commands, and VRAM can be especially limited on some systems). See element type *grating* for an example, or *cross* for an example of a type that needs some but not all three scripts.

Conserve time – *element_doShared*

element_doShared is a tool you can use to reduce the time used by multiple runs of a type script (multiple objects of the type) in an experiment. If you have a slow section of code, you can run it through *element_doShared* instead of directly. Then PsychBench will only run that code once across all objects of the type for the same inputs (or optionally index values—see *element_doShared* help). This can save a lot of time, e.g. if there are hundreds of similar objects across trials. Type *help element_doShared* for usage. ***element_doShared* has some overhead, so only use it for code that is slow.**

See for example script *grating_open2*.

1 If the user applies a staircase to any properties of the element, both *open1* and *open2* wait to run before the trial the object is in.

Conserve memory in *open2* – *element_setShared*

element_setShared is a tool you can use to reduce the memory used by multiple runs of an *open2* script (multiple objects of the type) in an experiment. If you want to set a large value to a property of *this*, you can set it using *element_setShared* instead of directly. Then PsychBench will only hold one copy of the value in memory across all objects of the type with the same index values (see *element_setShared* help). This can save a lot of memory, e.g. if there are hundreds of similar objects across trials. Type *help element_setShared* for usage. ***element_setShared* has some overhead, so only use it to set values that are large. You cannot use it for Psychtoolbox textures since it works on MATLAB values held in properties, and textures are held by Psychtoolbox elsewhere.**

See for example script *grating_open2*.

2.3. Showing a display – Texture method

The texture method for showing an object display is the usual approach for durable types. It can also be useful for quick types that need core display options (*this.position*, *.nn_eyes*, *.rotation*, etc.). In the texture method, make each image for the display centered on a texture (or off-screen window), and call *element_draw* or *element_redraw* in each iteration of *runFrame* to draw a texture to the experiment window to show in the next frame. This lets PsychBench apply all core display options for the object, so you don't need to. All the general approach and rules for showing a display still apply—see sec 1.10.

Making textures

The usual way to make a texture for an object display is using *element_openTexture*. This can make a blank texture (off-screen window) which you can draw to with any Psychtoolbox draw commands. It can also make a texture from an image matrix. *element_openTexture* automatically applies some core display options *this.channelResolution* and *.backColor*, default alpha blending / color mask. See help text for more information. See also the tool *element_setBackColor*, especially if you want to implement a transparent background.

You can also get textures containing images from Psychtoolbox functions like *Screen('GetMovieImage')*.

For a static display with one image, typically make the texture in the *open* (or *open3*) script to keep work out of frames. For a display with multiple images, you could make texture(s) there or in real time in *runFrame*.

Note for a dynamic display that you run by drawing an image to a blank texture in each iteration of *runFrame*, you can reuse the same texture. Use an optional input to *element_draw* to blank the texture for the next image (see example below).

You need to use Psychtoolbox *Screen('Close')* to close textures you open when done with them, generally either in *runFrame* or *close*. Note PsychBench automatically closes all textures if the experiment stops on error, so you don't need to write *catch* script code for that.

Drawing textures to the experiment window

element_draw/element_redraw

Use *element_draw* and/or *element_redraw* in *runFrame* to draw a texture to the experiment window to show in the next frame. They have the same usage and do the same thing except *_redraw* tells PsychBench that the display to draw is the same as the previous one the object drew. This just lets PsychBench do it more efficiently. So:

- Use *element_redraw* for a static display where you draw the same image and with the same inputs to *element_redraw* (if any) in most or all frames. You can only use it in frames where the image and inputs have not changed. However, note you **can** use it for the first image.
- Use *element_draw* for a dynamic display where you draw different images or with different inputs to *element_draw* in most or all frames. Or use it in frames where a static display changes.

See *element_draw* help text for more information, and examples below.

element_predraw

element_predraw is an optional function that lets PsychBench do some processing before the trial to reduce latency whenever the first call to *element_draw/redraw* happens for the object during the trial. If you have the first image for *element_draw/redraw* ready in the *open* (or *open3*) script, you can call *element_predraw* for it there. You must still call *element_draw/redraw* in *runFrame* as usual. The image and other inputs should be the same from *element_predraw* and the first call to *element_draw/redraw*. Most commonly use *element_predraw* for static displays (with *element_redraw* in *runFrame*).

Additive blending

By default *element_draw/redraw* draws textures to the experiment window using standard linear blend factors GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA. This means images occlude whatever is on screen, or blend linearly with it where they have transparency (alpha < 1). Optionally users can set core input property *this.addDisplay* = `true` to switch to additive blending for the object. Then *element_draw/redraw* draws textures using factors GL_SRC_ALPHA, GL_ONE. This means images add pixel-wise with whatever is on screen, weighted by image alpha. Usually this doesn't make sense for users to do. However, in cases where it does, your code can support it by reading *this.addDisplay* and branching to make images that will work additively. Often that means RGBA values that can be outside the range 0–1, both positive/negative (will add/subtract) and

centered at 0. To facilitate this, if *this.addDisplay* = `true` then *element_openTexture* makes floating point textures and pixel values are unclamped from 0–1. See element type *noise* for an example where users can specify additive noise.

Example 3. Texture method – Static display, Image from matrix

The object shows a picture from image data the user sets in an input property *data* (e.g. an *m*×*n* px ×3 RGB matrix). The user can also set display height in *height* (deg).

open: The *open* (or *open3*) script runs in the inter-trial interval before the trial the object is in. Here it converts type-specific input property *height* from deg or other distance units to px. It then uses *element_openTexture* to load the image matrix onto a texture. We save texture number to a record property for use later in *runFrame*. (If we used *open1*, *2*, *3* scripts, we would do at least the texture part in *open3* since VRAM can be especially limited on some systems—sec 2.2.)

```
this.height = element_deg2px(this.height);

data = this.data;
n_window = this.n_window;

n_texture = element_openTexture([], [], data);

this.n_texture = n_texture;
```

runFrame: *runFrame* repeats once each frame when the object is running. Here it draws the texture to the experiment window to show on screen next frame. It uses *element_redraw* since the image is the same in each frame. We apply *this.height* here by inputting it to *element_redraw* to scale the image to the specified height on screen. For precise timing we check core property *this.isEnding* to omit drawing in the last object frame so the display doesn't show in the frame after the object ends.

```
if ~this.isEnding
    height = this.height;
    n_texture = this.n_texture;

    this = element_redraw(this, n_texture, [], height);
end
```

close: The *close* script closes the texture using Psychtoolbox *Screen('Close')*.

```
Screen('Close', this.n_texture)
```

Example 4. Texture method – Dynamic display, Image from draw commands

This is a texture version of Example 2 in sec 1.10. The object shows a dot moving in a circle. The user can set parameters in input properties *radius* (deg), *velocity* (deg/sec), *dotSize* (deg), and *color* (RGB).

open: The *open* (or *open3*) script runs in the inter-trial interval before the trial the object is in. Here it converts type-specific input properties from deg (deg/sec for *velocity*) or other distance units to px. It then uses *element_openTexture* to make a blank texture which we will draw images to. We size the texture to fit the images, adding padding = diameter of the dot at the texture edge. We save texture number and center in record properties for use later in *runFrame*. By default *element_openTexture* fills the texture with an opaque color settable by the user (user default = experiment background color), and sets texture alpha blending and color mask appropriate to that. If you want something else you could use further inputs and/or *element_setBackColor*. (If we used *open1*, *2*, *3* scripts, we would do at least the texture part in *open3* since VRAM can be especially limited on some systems—sec 2.2.)

```
this.radius = element_deg2px(this.radius);
this.velocity = element_deg2px(this.velocity);
this.dotSize = element_deg2px(this.dotSize);

r = this.radius;
dotSize = this.dotSize;
n_window = this.n_window;

d = 2*r+2*dotSize;
textureSize = [d d];
n_texture = element_openTexture(textureSize);
textureCenter = (textureSize+1)/2;

this.n_texture = n_texture;
this.textureCenter = textureCenter;
```

runFrame: *runFrame* repeats once each frame when the object is running. Here we calculate dot position for next frame using expected time of next frame relative to object start. This position is relative to texture top left, with the display centered on the texture. We then use Psychtoolbox *Screen('DrawDots')* to draw the dot to our texture (note *DrawDots* needs position as a column vector). We then call *element_draw* to draw the texture to the experiment window to show on screen next frame. The last input to *element_draw* tells it to re-blank the texture so it's ready for the next iteration of *runFrame*. For precise timing we check core property *this.isEnding* to omit drawing in the last object frame so the display doesn't show in the frame after the object ends.

```
if ~this.isEnding
    r = this.radius;
    v = this.velocity;
    dotSize = this.dotSize;
    color = this.color;
    n_texture = this.n_texture;
    textureCenter = this.textureCenter;
    startTime = this.startTime;
    nextFrameTime = trial.nextFrameTime;

    t = nextFrameTime-startTime;
    a = v/r*t;
    p = [r*cos(a) r*sin(a)]+textureCenter;
    p = transpose(p);
    Screen('DrawDots', n_texture, p, dotSize, color)
    this = element_draw(this, n_texture, [], [], [], '-blank');
end
```

close: The *close* script just closes the texture using Psychtoolbox *Screen('Close')*.

```
Screen('Close', this.n_texture)
```

2.4. Object sleep/wake

typeOptions script variable *isSleepable* = `true`/`false` tells PsychBench whether objects of the type are "sleepable". Sleeping/Waking an object means PsychBench stops calling the *runFrame* script to stop it without permanently ending it, then later restarts calling *runFrame* to resume it. *isSleepable* just tells PsychBench whether *runFrame* code can accept this without breaking. If a user tries to set an object to sleep and *isSleepable* = `false`, they will get a helpful error message instead of a crash.

For most visual element types *runFrame* doesn't need any special code to be sleepable, so the default for *isSleepable* = `true`. If you need to edit *runFrame* to make objects sleepable, core properties *this.isStarting* and *.isEnding* are useful since they also apply (= `true`) in a frame where the object is about to wake/sleep respectively. If you need to check for wake/sleep specifically, you can use *this.isWaking* and *.isSleeping* instead.

2.5. Staircased properties

If an element is staircased that means the user told PsychBench to set one or more of its input properties when the trial containing the element runs, based on a staircase value that changes across trials (core input property *this.staircase* + a staircase object). Generally you don't need to do anything for this in element type code. PsychBench sets staircased properties before the trial, and runs all *open* scripts for the object after (footnote in sec 2.2). So in type scripts it looks the same as if the user set them.

2.6. Adjustable properties

Users can vary input properties of an element when it's running by listing it in property *vary*, which all elements have. Users can also let the subject adjust input properties using an adjuster element (e.g. Left key → +10). Both cases work the same in terms of element type code for the object being varied/adjusted:

In order for the user to use *vary* or adjustment, the target property must be ***adjustable***. To make a property adjustable, list it in *typeOptions* script variable *adjustable*. Any type-specific property can be adjustable, but only some core properties—see below. Aside from that, whether a property can be adjustable depends on your code: When a property is adjusted, PsychBench changes its value between frames (iterations of *runFrame*). Listing a property in *typeOptions* variable *adjustable* just

tells PsychBench that your *runFrame* script can accept and use such changes to the property at any time. (Note your *runFrame* code can change type-specific properties of *this* anytime. An "adjustment" is when the change is injected from outside.)

Here is the approach for an adjustable property in *runFrame*: In each iteration of *runFrame*, check *this.isAdjusted.<property>*. *isAdjusted* is a struct with a field for each adjustable property. A field = `true` if the property received an adjustment since the previous iteration of *runFrame*, else = `false`. If *this.isAdjusted.<property>* = `true`, do the following for the property:

(1) **Error check the new value:** Check the new value is acceptable and throw an error message for the user if not. e.g. for a property that must be ≥ 0, check it has not been adjusted < 0. Basically this checks that the user has set *vary* or the adjuster object to only apply valid adjustments. **Adjustments can't change data type (always numeric) or size, so you don't need to re-check those.**

(2) **Process the new value:** Adjusted property values are always in terms the user would set the property in (as if the user reset it during the experiment). This may not be the same as terms in element type code. The main example is a property users can set in deg or other distance units and the *open* script converts to px—each time the property is adjusted, its new value will be in the original units, so *runFrame* needs to re-convert it (sec 1.10 – *element_deg2px*).

(3) **Apply the new value:** Update the object's stimulus or functionality based on the change. Also update any dependent record properties. The previous value of the adjusted property is available in *this.prev.<property>* in case helpful.

Some or all of these steps may not be needed, depending on the property. If none is needed, you don't even need to check *isAdjusted* for the property. At the other end, some properties can't be made adjustable—for example, a property that is used in *open* in a way that is too slow to repeat in *runFrame*.

Note usually no more than one property is adjusted in a frame, but not necessarily. So generally you should check *this.isAdjusted.<property>* for each adjustable property. Also note a property can only be adjusted when the object is running, so you don't need to do anything outside *runFrame*. See also core property *this.propertyAdjustingNames*.

See example below.

Making core properties adjustable

You can make any type-specific properties adjustable. However, you can only make some core properties adjustable. Currently these are:

this.position
- *nn_eyes*
- *rotation*
- *colorMask*
- *alpha*
- *intensity*
- *contrastMult*

Handling an adjustable core property is the same as a type-specific property except PsychBench automatically handles error checking and all core functionality as usual (e.g. for *this.position*: converts it to relative to experiment window top left in pixels, applies it to move the object if you use the texture method for showing a display). Your code just needs to handle any type-specific functionality that also depends on the property (often nothing). **Either way, to make a core property adjustable you still need to list it in *typeOptions* script variable *adjustable*.**

Dependent record properties

When you list a property in *typeOptions* variable *adjustable*, you can also list "dependent" record properties. These are record properties that your *runFrame* script changes when the adjustable property changes. You don't need to list them in *adjustable* for *runFrame* to change them—it just affects how PsychBench shows them in experiment results output for users.

Documenting adjustable properties

At least for a durable type, if you make a property adjustable, add it to the list at the top of the user documentation text file.

Example

This adds to Example 2 in sec 1.10. The object shows a dot that moves in a circle. The user can set parameters in input properties *radius* (deg), *velocity* (deg/sec), *dotSize* (deg), and *color* (RGB). Here we add that input properties *velocity*, *color*, and *position* are adjustable.

typeOptions: In the *typeOptions* script we mark which input properties are adjustable. There are no dependent record properties for any of them.

```
adjustable = {
    'velocity'    []
    'color'       []
    'position'    []
    };
```

open: The *open* (or *open3*) script runs in the inter-trial interval before the trial the object is in. Here it converts type-specific input properties from deg (deg/sec for *velocity*) or other distance units to px. It also error checks all type-specific input properties. Adjustable properties are at their initial values here.

```
this.radius = element_deg2px(this.radius);
this.velocity = element_deg2px(this.velocity);
this.dotSize = element_deg2px(this.dotSize);

r = this.radius;
velocity = this.velocity;
dotSize = this.dotSize;
color = this.color;
n_window = this.n_window;

if ~(isOneNum(r) && r > 0)
    error('Property .radius must be a number > 0.')
end
if ~isOneNum(velocity)
    error('Property .velocity must be a number.')
end
if ~(isOneNum(dotSize) && dotSize > 0)
    error('Property .dotSize must be a number > 0.')
end
if ~isRgb1(color)
    error('Property .color must be a 1x3 vector with numbers between 0–1.')
end
```

runFrame: *runFrame* repeats once each frame when the object is running. Here it calculates dot position for next frame and draws the dot to the experiment window. In this example we add the `this.isAdjusted.<property>` blocks. For each adjustable property we check if it has received an adjustment since the previous frame. If yes, we do anything needed to check and handle the new value:

velocity: We convert the new value to px. No error checking is needed since adjustments can't change data type or size and *velocity* can be any number.

color: We error check the new value. We can check less than in *open* since adjustments can't change data type or size. No additional code is needed since our code just re-uses *color* in each frame.

position: Nothing is needed since this is a core property and our code just re-uses *position* in each frame.

```
if this.isAdjusted.velocity
    this.velocity = element_deg2px(this.velocity);
end
if this.isAdjusted.color
    color = this.color;

        if ~all(color >= 0 & color <= 1)
            error('Property .color numbers must be between 0-1.')
        end
end

if ~this.isEnding
    r = this.radius;
    v = this.velocity;
    dotSize = this.dotSize;
    color = this.color;
    startTime = this.startTime;
    position = this.position;
    n_window = this.n_window;
    nextFrameTime = trial.nextFrameTime;

    t = nextFrameTime-startTime;
    a = v/r*t;
    p = [r*cos(a) r*sin(a)]+position;
    p = transpose(p);
    Screen('DrawDots', n_window, p, dotSize, color)
end
```

2.7. Documentation

For a durable type you generally need to make user documentation. This needs to be a text file[2] called *<element type>.txt* in the type folder in order for the *pb* help command to find it. When you use *newPbType* it sets this up for you based on a template (you can redo this anytime by copying *doc template.txt* from *<PsychBench folder>/element type programming*). Documentation should include all input properties with their usage and default values, and record properties users might want to see in experiment results output.

▷ PsychBench documents all core properties at www.psychbench.org/docs, so you don't need to document them. The exception is if the element type uses one in a type-specific way.

pb menu heading and description

typeOptions script variables *heading* and *desc* are strings that are description and heading for the element type in the menu users get when they type *pb* at the command line. You should set a one-liner description. Leaving heading at automatic `[]` puts the type under "Visual", which is generally what you want.

2 PsychBench handles different line break conventions in Windows/Mac text files automatically.

2.8. Name conventions

For a durable type it's good to follow a few PsychBench conventions for property/variable names:

- Names are in lower camel case, e.g. *linearDotMask*, *brightness*.
- Numeric labels for things (as opposed to the thing itself) start with *n_*, *i_*, or similar, e.g. *n_frame*, *i_file*. For multiple, start with *nn_* or similar, e.g. *nn_frames*.
  (For numbers that are not labels, no need for this.)
- Text labels for things end with *Name* or similar, e.g. *fileName*. For multiple, end with *Names* or similar, e.g. *fileNames*.
  (For text that isn't a label, no need for this.)
- Quantities start with *num*, e.g. *numResponses*.

3. Core property reference

This section highlights core properties that can be useful in element type code. These include both core input properties which can be set by users, and core record properties which PsychBench sets during experiments. **Read only as needed.**

3.1. Core properties – Visual element objects (*this.<property>*)

position
nn_eyes
rotation
flipHorz
flipVert
colorMask
alpha
intensity
contrastMult
convolution
shader
filterOrder
filterGrayscale
filterResolution
filterGamma
channelResolution
backColor
addDisplay

Core input properties setting options for the object display. See www.psychbench.org/docs/elementwithscreen. If you use the direct draw method for showing an object display, your code can read and apply these properties or leave any of them unimplemented (sec 1.10). If you use the texture method, PsychBench applies all of them, so you don't have to (sec 2.3).

Specific notes:

***position*: For users when making experiments, *position* has `[0 0]` = experiment window center and default units deg. However, in element type code it has `[0 0]` = window top left and units px to facilitate use with Psychtoolbox functions.**

backColor: In element type code this always a 1×4 RGBA vector that is the current object background color, e,g, including whatever it defaulted to or whatever you have set with *element_setBackColor* if the user didn't set *backColor*. If you need to know what the user actually set, that is available in *this.user.backColor*.

(*depth*: PsychBench always applies this property (including in direct draw) since it works across objects, layering them relative to each other.)

n_window

Psychtoolbox window number (pointer/handle) for the on-screen experiment window. You can use this with *Screen* draw functions in the direct draw method for showing an object display (sec 1.10). Also available in *devices.screen.n_window*.

displayRect

Approximate object display position and size in a 1×4 vector `[x_tl y_tl x_br y_br]` relative to experiment window top left (px). Automatic based on texture size in the texture method (sec 2.3). Not available in the direct draw method (sec 1.10).

isStarting
isEnding
isWaking
isSleeping

isStarting and *isEnding* = `true` in the *runFrame* script if it's in object frame 0 (first iteration) or last object frame (last iteration) respectively, else = `false`. You can check these to branch to different code or exclude code when the object is starting or ending. See sec 1.7.

isStarting and *isEnding* also = `true` in a frame where the object is about to wake/sleep respectively (sec 2.4). If you need to check for wake/sleep specifically, you can use *this.isWaking* and *.isSleeping* instead.

startTime
endTime
duration

In experiment results output, *startTime* and *endTime* are relative to trial 1 start. However, in element type code they are in system time to facilitate use with Psychtoolbox functions.

startTime is object start time (system time, sec). By default PsychBench measures start time as object frame 1 start (after the first iteration of *runFrame* in frame 0). Available in all iterations of *runFrame* and in *close*.

endTime is object end time (system time, sec) and *duration* is end time – start time. By default PsychBench measures end time as last object frame end (after the last iteration of *runFrame*). Available in the last iteration of *runFrame* and in *close*.

Expected/Actual times: During *runFrame*, these properties contain expected frame transition times. i.e. they are times that the frame transition *should* happen or *should have* happened, not precisely when it did happed (e.g. if dropped frames occurred). This allows you to use *startTime* in object frame 0 before start has occurred, and then maintain the object's expected time course in later frames (for example, to maintain synchronization with other objects the user set to start at the same time). However, when users see these properties in experiment results output, they want actual start and end times. PsychBench replaces *this.startTime/endTime* with actual measured frame transition times before the *close* script. So if your code needs to use these properties for purposes related to experiment results output, wait until *close*.

isWaiting
ran

`true/false`: whether the object is still waiting to start (no iterations of the *runFrame* script have run), and ran and ended, respectively. These can be useful in *close* and *catch* scripts. e.g. you can check `this.ran` in *close* to branch to code if the object ran, or check `~this.isWaiting` in *catch* to branch to code if the object started in the experiment by the time the error occurred.

isAdjusted.<property>
prev.<property>
user.<property>
propertyAdjustingNames

If you make any input properties adjustable, PsychBench adds core properties *isAdjusted*, *prev*, and *user*. These are further structs with field names matching adjustable property names:

isAdjusted.<property> = `true` in the *runFrame* script if a property received an adjustment since the previous frame, else = `false`.

prev.<property> contains the property's previous value (or `[]` if no adjustment yet), in case helpful.

user.<property> contains the current value of the property in terms the user would set (e.g. in deg, not px), in case helpful. The current value in terms used in element type code is in the property itself, which generally you will use for all purposes.

Lastly *propertyAdjustingNames* is a cell array of strings that are names of properties that could receive adjustments when the object is running.

See sec 2.6.

propertySetNames

A cell array of strings that are names of input properties the user set, as opposed to left at default. (Maybe useful since properties left at default will have their default values in element type code, same as if the user set them to those values.)

3.2. Core properties – Trial objects (*trial.<property>*)

backColor

A 1×3 RGB vector with numbers between 0–1 that is background color for the trial.

nextFrameTime
nextnextFrameTime

trial.nextFrameTime is expected mid time of next frame (system time, sec) in the *runFrame* script. If precise timing is important, this is the measure of "current time" to use when cueing functionality that will occur next frame, including generating the next image in a dynamic display.

trial.nextnextFrameTime is like *nextFrameTime* except it's for the frame after next frame, so two frames forward. Not used as often.

frameStartTimes
frameMidTimes

If you need frame start or mid times more generally they are available in these properties in the *runFrame* script. Each is a 1×5 vector of times (system time, sec) for:

(1) 2 frames backward
(2) 1 frame backward (previous frame)
(3) current frame
(4) 1 frame forward (next frame)
(5) 2 frames forward

startTime
endTime
duration

In experiment results output, *startTime* and *endTime* are relative to trial 1 start. However, in element type code they are in system time to facilitate use with Psychtoolbox functions.

Trial start and end time (system time, sec) and duration (sec). Available in the *close* script.

3.3. Core properties – Experiment object (*experiment.<property>*)

frameRate
frameInterval

Nominal frame rate (frames/sec) and interval (sec). Just = the screen's refresh rate.

startTime

Experiment start time (system time, sec). Available in the *close* script.

3.4. Core properties – Screen objects (*devices.screen.<property>*)

n_window

Psychtoolbox window number (handle/pointer) for the on-screen experiment window. Same as *this.n_window*.

windowSize
windowCenter

Experiment window size `[width height]` and center `[x y]` relative to its top left (px).

refreshRate
refreshInterval

Screen refresh rate (refreshes/sec) and interval (sec). Also available in *experiment.frameRate*, *.frameInterval*.

px2fontSize

Font size set using Psychtoolbox *Screen('TextSize')* can be inconsistent across systems, at least if the system doesn't use the high quality text renderer. *px2fontSize* is a scale factor that PsychBench calibrates to fix this. Multiply the font size you want (px) by *px2fontSize* to get the font size you should submit to *Screen('TextSize')* (also px).