# Chapter 10
# Psychtoolbox: Sound, Keyboard and Mouse

*PTB has a number of functions that can be useful to program behavioral experiments. Although their number is high, there is a relatively small number of core functions that we need to know to program a large spectrum of experiments. These core functions are presented in this chapter.*

## Timing

PTB has many functions dedicated to timing issues; probably the simplest one is `WaitSecs()`, which waits the number of seconds specified in the input argument. `WaitSecs()` can be used to set the pauses within trials (or blocks of trials) of your experiment. By running the following script, the monitor will remain white for 10 s before returning to its normal appearance.

**Listing 10.1**

```
try
    Screen('OpenWindow', 0);
    WaitSecs(10);
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Another useful function to manage timing is `GetSecs()`, which returns the time (in seconds) elapsed between when you switched on the computer and when

M. Borgo et al., *MATLAB for Psychologists*,
DOI 10.1007/978-1-4614-2197-9_10, © Springer Science+Business Media, LLC 2012

`GetSecs()` has been called. In the following example, code listing 10.1 is extended to calculate the time elapsed between the two `GetSecs()` calls.

**Listing 10.2**

```
try
    Screen('OpenWindow', 0);
    t0 = GetSecs;
    WaitSecs(5);
    t1 = GetSecs;
    Screen('CloseAll');
    t_elapsed = t1 - t0;
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

`GetSecs()` is useful in several contexts, for example when it is used together with the functions controlling the keyboard and the mouse.

## Priority

When we are running experiments, we want to allocate all the computer's resources (e.g., memory and CPU) for the experiment only. PTB lets you do this thanks to the priority functions. You should know that when you use a computer, although you may have only one application open and visible on the monitor (e.g., MATLAB), there are several applications running in the background. All these applications use the CPU and computer memory, and therefore they reduce the available resources. This might be a problem if we are interested in getting the exact time a participant in our experiment has pressed a button. PTB allows for allocating the maximal priority to the event we want. However, this maximal priority can be kept for only a few seconds. For this reason it is better to get it just before calling this event and then setting the computer priority back to normal. The levels of priority are identified by integers, which depend on the operating system. To find out the number corresponding to the maximum priority level in your system, use the MaxPriority function. The following script shows its use.

**Listing 10.3**

```
try
    whichscreen = max(Screen('Screens'));
    [myscreen, screen_rect] = Screen('OpenWindow', whichscreen);
    Screen('FillRect', myscreen, 0, CenterRect([0, 0, 100, 100],
screen_rect));
    priorityLevel=MaxPriority(window);
    Priority(priorityLevel);
    t0 = Screen('Flip', myscreen);
    Screen('Flip', myscreen, t0 + 1);
    Priority(0);
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Let us analyze the script. Before calling the flip subfunction, we have interrogated the system about the maximal priority available and stored the returned value in the `priorityLevel` variable. Then, we set the priority to its maximum level, and as soon as the flip is over we set the priority back to zero. Zero is the default priority that is normally attributed to all applications running on the computer. In other words, during normal usage, the priority of all application is equal to zero.

## Sound Functions

PTB includes also some functions that can be used for synthesizing and playing sounds. These functions are particularly suitable for psychological experiments because they use drivers that are highly time-efficient in comparison to the native sound drivers of Windows or those of other operating systems. The main sound function in PTB is the `PsychPortAudio` function. This function can work both synchronically and asynchronically. The way you call this function is similar to the `Screen` function. The function name must be followed by the subfunction name and by a variable list of parameters that varies according to the subfunction. In the following table we present the most important `PsychPortAudio` subfunctions. In the table, the input parameters given within square brackets are optional:

| Sub Function | Command | Explanation |
|---|---|---|
| Version | `struct=PsychPortAudio ('Version')` | `return the version of PsychPortAudio in a struct` |
| Verbosity | `oldlevel=PsychPortAudio ('Verbosity' [,level]);` | `Set level of verbosity for error/warning/ status messages` |

(continued)

(continued)

| Sub Function | Command | Explanation |
|---|---|---|
| GetOpenDeviceCount | `count=PsychPortAudio ('GetOpenDeviceCount');` | `Return the number of currently open audio devices` |
| Open | `pahandle=PsychPortAudio('Open' [, deviceid][, mode][, reqlatencyclass][, freq][, channels] [, buffersize][, suggestedLatency][, selectchannels]);` | Open a PortAudio audio device and initialize it. Returns a 'pahandle' device handle for the device |
| Close | `PsychPortAudio('Close' [, pahandle]);` | Close a PortAudio audio device |
| FillBuffer | `[underflow, nextSampleStart Index, nextSampleETASecs]=PsychPortAudio('FillBuffer', pahandle, bufferdata [, streamingrefill=0][, startIndex=Append]);` | Fill audio data playback buffer of a PortAudio audio device. 'pahandle' is the handle of the device whose buffer is to be filled |
| CreateBuffer | `bufferhandle=PsychPortAudio('CreateBuffer' [, pahandle], bufferdata);` | Create a new dynamic audio data playback buffer for a PortAudio audio device and fill it with initial data |
| DeleteBuffer | `result=PsychPortAudio('DeleteBuffer'[, bufferhandle] [, waitmode]);` | Delete an existing dynamic audio data playback buffer |
| Start | `startTime=PsychPortAudio ('Start', pahandle [, repetitions=1] [, when=0] [, waitForStart=0] [, stopTime=inf]);` | Start a PortAudio audio device |
| GetStatus | `status=PsychPortAudio ('GetStatus', pahandle);` | Returns 'status', a struct with status information about the current state of device 'pahandle' |
| Stop | `[startTime endPositionSecs xruns estStopTime]= PsychPortAudio('Stop', pahandle [, waitForEndOfPlayback=0] [, blockUntilStopped=1] [, repetitions] [, stopTime]);` | Stop a PortAudio audio device. The 'pahandle' is the handle of the device to stop |

The help of each subfunction can be viewed in the same way you can view the help of the Screen function, i.e., by typing a question mark at the end of the subfunction name (e.g., `PsychPortAudio('Open?')`). For an overview of the function, type `PsychPortAudio()` or "`help PsychPortAudio`".

The usage of `PsychPortAudio` is the following. First, you need to open the audio device. Second, you need to fill a sound buffer with your sound. Third, you need to play the sound. Moreover, PTB developers suggest that you call

`InitializePsychSound` before the first invocation of `PsychPortAudio`. If you omit this call, the initialization of the driver may fail, and MATLAB may return some "`Invalid MEX file`" error.

Here we present an example showing another useful sound function: `MakeBeep()`. `Makebeep` synthesizes a pure tone of a given frequency, duration, and sample rate. Note that in the following example, `PsychPortAudio` works synchronically. To effect this, we use the subfunction "`Stop`", which has an optional parameter. This parameter enables us to specify when to stop the beep's playback. Therefore, in the example this parameter is set equal to "`d`", i.e., the overall duration of the beep.

**Listing 10.4**

```
try
    f = 1000; % frequency in Hz
    d = 1; % duration in sec
    sr = 44100; % sample rate in Hz
    beep = MakeBeep(f, d, sr); %synthesis of the beep
    InitializePsychSound;
    pahandle = PsychPortAudio('Open', [], [], [], sr, 1);
    PsychPortAudio('FillBuffer', pahandle, beep);
    PsychPortAudio('Start', pahandle);
    PsychPortAudio('Stop', pahandle, d);
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

## Getting Participants' Inputs: Keyboard and Mouse Functions

When we run behavioral experiments we usually collect responses from our participants. In the majority of these experiments the response is collected through either the keyboard or the mouse.

### *Keyboard Response*

There are two classes of functions for capturing keyboard events. The first class is *keypress*-oriented. The second is *character*-oriented. The former fulfill the majority of an experimental psychologist's needs; hence we describe this set of functions only. `Kbwait()` and `KbCheck()` are the main keypress-oriented functions. They work in a similar way; however, `KbWait` waits for user input, whereas `KbCheck` does not. In other words, `KbWait` stops the script until the user presses a key on the

keyboard, whereas `KbCheck` checks whether a key-press event has occurred when the function is called. Therefore, if at that moment no key is been pressed, the script continues. Because of the different characteristics of these functions we recommend using `KbCheck` for collecting responses such as high-accuracy response times, and to use `KbWait` for other kinds of responses.

## *"Press Any Key to Proceed"*

In many circumstances, we need the participant to press a key to proceed with the experiment. For example, the key press can follow the presentation of instructions. `KbWait` can be used in such circumstances when it is called with no input argument.

**Listing 10.5**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
    [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS ANY KEY TO PROCEED','center',
'center');
    Screen('Flip', myscreen);
    KbWait;
    screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

## *"Press the Spacebar to Proceed"*

In other circumstances, we want the participant to press a specific key to proceed. For example, we may want the participant to press the spacebar to go further with the experiment. To do this, we need to know something more about the keyboard. Both `KbWait` and `KbCheck` return as output the argument `keycode`. `keyCode` is a 256-element array in which every key is mapped to a number. The key that has been pressed is identified by the fact that its corresponding position in the array turns from the default 0 to 1. For example, in the Mac OS, the "return" key is mapped to the 40th position and the spacebar to the 44th. In contrast, the same keys are mapped to positions 13 and 32, respectively, under Windows. To know the position of a

specific key in the array, use the KbName function. The following table outlines this function:

| Usage | Explanation |
|---|---|
| KbName('s') | Return the keycode of the indicated key (inputted as a string). Special keys such as spacebar, return, and so on, are also passed as a string (e.g., 'space', 'return') |
| KbName(keyCode) | Return the label of the key identified by keyCode |
| KbName | Waits 1 s and then calls KbCheck. KbName then returns a cell array holding the names of all keys that were down at the time of the KbCheck call |
| KbName('KeyNames') | Print out a table of all keycodes->keynames mappings |
| KbName('KeyNamesOSX') | Print out a table of all keycodes->keynames mappings for MacOS-X |
| KbName('KeyNamesOS9') | Print out a table of all keycodes->keynames mappings for MacOS-9 |
| KbName('KeyNamesWindows') | Print out a table of all keycodes->keynames mappings for MS-Windows |
| KbName('KeyNamesLinux') | Print out a table of all keycodes->keynames mappings for GNU-Linux, X11 |

Now that we have mastered keyboard events, let's deal with the case in which the program waits the participant to press the spacebar.

**Listing 10.6**

```
try
    screens = Screen('Screens);
    whichscreen = max(screens);
    [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS THE SPACEBAR TO PROCEED', 'center',
'center');
    Screen('Flip', myscreen);
    space = KbName('space');
    [secs, keyCode] = KbWait;
    while keyCode(space)==0
       [secs, keyCode] = KbWait;
    end
    screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

The core of this script lies in the while loop. Before the loop we check the keyboard. Furthermore, we store the key pressed by the participant in the variable keyCode. As long as the participant presses any key on the keyboard other than the spacebar (or presses no key at all), the while loop is repeated. However, as soon as the participant presses the spacebar, the loop quits. In the script, the number corresponding to the spacebar is taken using KbName. Because keyCode is the second output argument returned by KbWait, we need to save the secs argument first.

### *"Press Any Key to Respond"*

In many tasks, we ask the participants to produce a binary response. The following example extends the previous one to allow for a binary response.

**Listing 10.7**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
     [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS ANY KEY TO PROCEED', 'center',
'center');
    Screen('Flip', myscreen);
    KbWait;
    y = KbName('y');
    n = KbName('n');
    words={'home', 'bank', 'nose', 'dog', 'car'};
    responses = zeros(length(words), 1);
    for i=1:length(words)
        WaitSecs(1);
        DrawFormattedText(myscreen, char(words(i)), 'center', 'center');
        Screen('Flip', myscreen);
        [secs, keyCode] = KbWait;
        while keyCode(y)==0 && keyCode(n)==0
            [secs, keyCode] = KbWait;
        end
        if keyCode(y) == 1;
            responses(i) = 1;
        end
    end
    screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

This script first opens a PTB window with '`OpenWindow`', then echoes "`press any key to proceed`". After the participant presses any key, the function `KbName` gets the "`y`" and "`n`" key progressive numbers. Next, we present the stimulus (the words) within a for loop. Note that we use `WaitSecs` to pause the experiment when passing from one trial to the next. Then we check the keyboard within the for loop, by means of a while loop. The check is done as in the previous example, with the difference that by means of the AND logical condition, the program stops until the participant presses either "`y`" or "`n`". Finally, we store the response in a numeric array in which 1 stands for y and 0 for n. This might be useful for later statistical analysis.

### *Reaction-Time Detection*

The simplest reaction time to implement is that of detection (aka simple reaction time). In the detection reaction time, the participant has to press a key as soon as s/he

detects something (e.g., a visual stimulus is presented on the screen). We recommend using `KbCheck` rather than `KbWait`. This is because `KbWait` checks the keyboard every 5 ms. Therefore, it adds an 5 extra ms of uncertainty to measurements. The same problem does not occur with `KbCheck`.

In the following example, we ask the participant to press the spacebar as soon as a stimulus appears on the screen.

**Listing 10.8**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
     [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS ANY KEY TO PROCEED', 'center',
'center');
    Screen('Flip', myscreen);
    KbWait;
    space = KbName('space');
    ntrials = 5;
    rt = zeros(ntrials, 1);
    stimulus duration = zeros(ntrials, 1);
    for i=1:ntrials
        WaitSecs(1);
        Screen('FrameOval', myscreen, 0, CenterRect([0, 0, 10, 10], rect));
        onset_fixation = Screen('Flip', myscreen);
        Screen('FillRect', myscreen, [255, 0, 0], CenterRect([0, 0, 50,
50], rect));
        random_delay = rand+0.5;
        onset_stimulus = Screen('Flip', myscreen, onset_fixation +
random_delay);
        t0 = GetSecs;
        [keyIsDown, secs, keyCode] = KbCheck;
        while keyCode(space)==0
            [keyIsDown, secs, keyCode] = KbCheck;
        end
        rt(i) = secs - t0;
        stimulus_offsettime = Screen('Flip', myscreen);
        stimulus_duration(i) = stimulus_offsettime - t0;
    end
    screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

This script is similar to the previous examples in that it waits until the a keypress event occurs. We then run five trials in a `for loop`. In each trial, the program first waits for 1 s, then, by means of the `FrameOval` subfunction, a fixation point appears. Then a red square, the target stimulus, appears at a random time interval after the fixation point has disappeared. As soon the target stimulus flips, `GetSecs` stores the onset time of the stimulus in the `t0` variable. `KbCheck` checks the keyboard until the participant presses the spacebar. At that time, the reaction time is stored. It should be kept in mind that `KbCheck` registers the time in seconds from when the computer has been switched on. Hence, to get the actual reaction time we need to calculate the difference between `secs` and `t0`, i.e., the difference between

the moment the spacebar has been pressed and the stimulus onset. Finally, note that when calling `KbCheck` we collect also the variable `keyIsDown`. `keyIsDown` is the first output argument of `KbCheck`, and it is a logical value (i.e., 0–1) that is equal to 1 when the user presses one key at the moment `KbCheck` is called.

## *Choice Reaction Time*

A second kind of reaction time is called *choice*. The participant has to press one button in response to a particular stimulus and another button in response to another stimulus. In the following example, the participant has to press "`r`" for a red and "`g`" for a green square. Furthermore, we save the participant's response to check its accuracy.

**Listing 10.9**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
    [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS ANY KEY TO PROCEED', 'center',
'center');
    Screen('Flip', myscreen);
    KbWait;
    g = KbName('g');
    r = KbName('r');
    ntrials = 6;
    colorsequence = [1, 2, 2, 2, 1, 1];
    rt = zeros(ntrials, 1);
    response = zeros(ntrials, 1);
    stimulus_duration = zeros(ntrials, 1);
    for i=1:ntrials
        WaitSecs(1);
        Screen('FrameOval', myscreen, 0, CenterRect([0, 0, 10, 10], rect));
        onset_fixation = Screen('Flip', myscreen);
        if colorsequence(i) == 1
            stimuluscolor = [255, 0, 0];
        else
            stimuluscolor = [0, 255, 0];
        end
        Screen('FillRect', myscreen, stimuluscolor, CenterRect([0, 0, 50,
50], rect));
        random_delay = rand+0.5;
        onset_stimulus = Screen('Flip', myscreen, onset_fixation +
random_delay);
        t0 = GetSecs;
        [keyIsDown, secs, keyCode] = KbCheck;
        while keyCode(g)==0 && keyCode(r)==0
            [keyIsDown, secs, keyCode] = KbCheck;
        end
        rt(i) = secs - t0;
        response(i) = find(keyCode==1);
        stimulus_offsettime = Screen('Flip', myscreen);
        stimulus_duration(i) = stimulus_offsettime - t0;
    end
    screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

As a difference from the previous script, here the color of the square has to be monitored trial by trial; hence we save the color in the `colorsequence` variable. Moreover, we declare the `response` variable where we store the response of the participant. In the `if` function, embedded in the for loop, the color of the square that will appear in the trial is set. Finally, in the while loop that collects the participant's response we include as valid response both the "`r`" and the "`g`" keys. When the response is collected, we store both the reaction time, as in the previous example, and the key that has been pressed by the participant.

## *Go/No-Go Reaction Time*

In some cases, a participant has to react selectively to different stimuli. The following example shows the go/no-go reaction-time paradigm by modifying the previous example. The participant's task is to press the spacebar when a red square appears and to do nothing when a green square appears.

**Listing 10.10**

```
screens = Screen('Screens');
whichscreen = max(screens);
[myscreen,rect]=Screen('OpenWindow',whichscreen);
DrawFormattedText(myscreen, 'PRESS ANY KEY TO PROCEED', 'center', 'center');
Screen('Flip', myscreen);
KbWait;
space = KbName('space');
ntrials = 6;
colorsequence = [1, 2, 2, 2, 1, 1];
rt = zeros(ntrials, 1);
stimulus_duration = zeros(ntrials, 1);
for i=1:ntrials
    WaitSecs(1);
    Screen('FrameOval', myscreen, 0, CenterRect([0, 0, 10, 10], rect));
    onset_fixation = Screen('Flip', myscreen);
    if colorsequence(i) == 1
        stimuluscolor = [255, 0, 0];
    else
        stimuluscolor = [0, 255, 0];
    end
    Screen('FillRect', myscreen, stimuluscolor, CenterRect([0, 0, 50, 50],
rect));
    random_delay = rand+0.5;
    onset_stimulus = Screen('Flip', myscreen, onset_fixation + random_delay);
    t0 = GetSecs;
    [keyIsDown, secs, keyCode] = KbCheck;
    while keyCode(space)==0 && (secs - onset_stimulus) < 2
        [keyIsDown, secs, keyCode] = KbCheck;
    end
    rt(i) = secs - t0;
    stimulus_offsettime = Screen('Flip', myscreen);
    stimulus_duration(i) = stimulus_offsettime - t0;
end
screen('CloseAll');
```

The main difference between this example and the previous one is the condition that has to be satisfied to exit from the while loop in which the response is collected. Here, the while loop is exited in two cases: either when the user presses the spacebar

or when the stimulus stays on screen for more than the 2 s. This second condition is controlled by the `onset_stimulus` and `secs` variables. The first variable is the square onset time; the second is a time value that continuously updates every time the keyboard is checked.

### *Reaction Times Within a Video Clip*

So far, we have seen how to collect a reaction time for static stimuli. How can we collect reaction times when a video clip is being played? This is a particular case, and it requires a different technique to collect the reaction time. Indeed, in previous examples, the `while` loop where `kbCheck` gets the timing stopped the execution of any other command. Therefore, if we were inserting the while loop within a `for` cycle used to create a video clip, we would stop the clip until a key is pressed. To solve this problem we need to call `KbCheck` once every refresh cycle. The limit of this approach is that the accuracy of the reaction time is linked to the refresh rate: the higher the refresh rate, the greater the accuracy of the response time. Of course, we need to call `GetSecs` just before showing the video clip so that the reaction is calculated as a difference between the motor reaction and the moment the video starts. The following example shows how to do it.

**Listing 10.11**

```
t0 = GetSecs;
for i = 1:number_of_frames
    % draw the video clip
    % flip the video clip
    [keyIsDown, t_press, keyCode] = KbCheck;
    if keyCode(responsekey) == 1
        t_press = t_press-t0;
    end
end
```

However, this technique has a problem. When the participant presses the key, the participant' s finger stays on the key for a certain time, and obviously the key-touch is not instantaneous. We do not exactly know the duration of this time, but let's assume that the finger stays on the key for about 50 ms. Let's suppose we are working at a refresh rate of 100 Hz and therefore `KbCheck` checks the keyboard every 10 ms. When the user first presses the response key, the reaction time is calculated. Then, the next iteration of the for loop occurs. Because the finger is still on the key, the reaction time is calculated twice, and then there is a new iteration of the for loop; the video clip continues, and the finger is still on the key and the reaction time is calculated once again, and so on. In practice, if we were using the script above we would be calculating the reaction time based on the key-release motor action instead

of the key strike motor action. Therefore, when you are measuring the reaction time within a video clip, the conditional `if` has to be written differently:

**Listing 10.12**

```
t0 = GetSecs;
firsttouch = 0;
for i = 1:number_of_frames
    % draw the movie
    % flip the movie
    [keyIsDown, t_press, keyCode] = KbCheck;
    if keyCode(responsekey) == 1 && firsttouch == 0
        t_press = t_press-t0;
        firsttouch = 1;
    end
end
```

In the example, we added the variable `firsttouch`, which is set to `0` and becomes equal to `1` when the participant first touches the keyboard. In this way, in the following `for` loop the response time is not recalculated because the variable `firsttouch` is now `1`.

Now let's combine everything into a working example. A disc is placed in the middle of the screen and the participant has to detect when the disc starts its motion. In order to prevent anticipations due to fixed timing, the start of the disc's motion is controlled by a random parameter.

**Listing 10.13**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
    [w, rect] =Screen('Openwindow', whichscreen);
    disc = CenterRect([0, 0, 20, 20], rect);
    Screen('FillOval', w, 0, disc);
    Screen('Flip', w);
    space = KbName('space');
    firsttouch = 0;
    WaitSecs(1)+rand*2;
    t0 = GetSecs;
    for i = 1:200
        Screen('FillOval', w, 0, [disc(1)+i, disc(2), disc(3)+i, disc(4)]);
        Screen('Flip', w);
        [keyIsDown, secs, keyCode] = KbCheck;
        if keyCode(space) == 1 && firsttouch == 0
            rt = secs-t0;
            firsttouch = 1;
        end
    end
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

## *Mouse Input*

The mouse is not used as often as the keyboard to collect the participant's response. However, PTB is provided with functions enabling for this possibility. Before describing these functions, let us present two very simple (but extremely useful) functions: `HideCursor` and `ShowCursor`. These functions can be called with no input argument. They hide the mouse pointer and show it, respectively. Here is an example of how to use them.

**Listing 10.14**

```
try
    Screen('Openwindow', 0);
    HideCursor;
    WaitSecs(5);
    ShowCursor;
    WaitSecs(5);
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

The most important functions contained in PTB to deal with the mouse are `GetMouse`, `GetClick`, and `SetMouse`. `SetMouse` places the mouse cursor at the desired x and y coordinates. Therefore, the function waits for at least two input parameters (i.e., the desired x and y positions of the mouse). A third optional parameter can be passed to the function which is a screen pointer. In the current example, the mouse cursor is placed in the middle of the screen every 2 s. Try to move the mouse while the example is running:

**Listing 10.15**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
    [mainscreen, rect] = Screen('Openwindow', whichscreen);
    for i = 1:5
        SetMouse(rect(3)/2, rect(4)/2, mainscreen);
        WaitSecs(2);
    end
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

`GetMouse()` returns three arguments: The firsts two are the `x` and `y` mouse coordinates in pixels. The third is a logical vector whose length corresponds to the number of mouse buttons. When a button is pressed, the corresponding bit in the vector is set to 1, so it is easy to start a procedure when a specific button is pressed. This function receives, as optional argument, the pointer to the screen (in case you have more than one screen).

`GetClicks()` is similar to `GetMouse()` and takes three arguments. The first is the number of mouse clicks that the user performed within a time interval. The time interval is set by the variable `ptb_mouseclick_timeout`. The other two are the x and y current positions in pixel coordinates, respectively, of the cursor position when the first click has been executed.

In the following example we capture the position of the mouse click with `GetClicks`, and each time we display the x-y coordinates of the click.

**Listing 10.16**

```
try
    screens = Screen('Screens');
    whichscreen = max(screens);
     [myscreen,rect]=Screen('OpenWindow',whichscreen);
    DrawFormattedText(myscreen, 'PRESS THE SPACEBAR TO PROCEED', ...
                      'center', 'center');
    Screen('Flip', myscreen);
    KbWait;
    Screen('Flip', myscreen);
    for i = 1:5
        [clicks, x, y, whichButton] = GetClicks;
        while ~clicks
            [clicks, x, y, whichButton] = GetClicks;
        end
        Screen('DrawText', myscreen,sprintf('This is the coordinate: X=%i,
Y=%i ', x, y), x, y, 0);
        Screen('Flip', myscreen);
    end
    Screen('CloseAll');
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Note that `GetClicks` is called in a similar way to `KbCheck`. The function returns four output arguments. The first is a logical value informing whether any click occurred. We call `GetClicks` with a while loop as well as `KbCheck`. Here, however, we proceed (i.e., the program continues) as soon as the user presses the mouse.

## Using Participants' Input to Manipulate Shape Characteristics

In chapter 9 we saw how to design simple figures using PTB, how to write text into the destination window, while in this chapter we have seen how to capture participant input. The aim of this section is to combine these things to allow the user to

manipulate the characteristics of shapes. This may be useful, for example, when you adopt the adjustment method as the psychophysics method for your experiments. In this case, you want the participants to adjust a shape characteristic to match the same characteristic of another shape. The following code listing shows how to use the participants' inputs to manipulate the color of a rectangle.

### *Keyboard Manipulations*

As stated in Chap. 6, the simultaneous lightness contrast is probably the most studied phenomenon in lightness perception [see Kingdom (1997) for a historical review]. Listing 10.17 shows how to measure, in RGB values, this phenomenon through the adjustment method using the keyboard.

**Listing 10.17**

```
try
   HideCursor;
   [w, rect] = Screen('Openwindow',0);
   grey= 128;
   black =0;
   adjustable = 90;
   cx = rect(3)/2;
   cy = rect(4)/2;
   size=60;
   displacement = cx/2;
   coordLeft =[cx-displacement-size cy-size cx-displacement+size cy+size];
   coordRight=[cx+displacement-size cy-size cx+displacement+size cy+size];
   coordBlack = [0 0 cx rect(4)];
   Screen('TextSize', w,12) ;
   Instructions = 'Press the Up and Down arrow keys to adjust the' +
   'of the square to your right to match the color of the other.';
   Instructions2 = 'Press Esc when you are satisfied with your match.';
   while 1
      Screen('FillRect',w,  black,coordBlack);
      Screen('DrawText',w, Instructions, 10, 20,grey);
      Screen('DrawText',w, Instructions2, 10,40,grey);
      Screen('FillRect',w,  grey,coordLeft);
      Screen('FillRect',w,  adjustable,coordRight);
      Screen('Flip', w);
      [ keyIsDown, s, keyCode ] = KbCheck;
      if keyCode(38)
         adjustable=adjustable+1;
      elseif keyCode(40)
         adjustable=adjustable-1;
      elseif keyCode(27)
         break;
      end
   end
   Screen('DrawText',w, sprintf('Your final RGB was %d, press any key to
exit.', adjustable),10,60);
   Screen('Flip', w);
   while kbcheck end
      KbWait;
      ShowCursor;
      Screen('Close',w)
catch
   screen('Close',w)
   rethrow(lasterror)
end
```

**Analysis**

Line 2 hides the cursor to avoid any unwanted interference.

Lines 4–6 implement the variables for `color` They are scalar because only achromatic colors will be used. The variable `adjustable` is the color for the adjustable patch.

Lines 7–13 implement variables for the positioning and coordinates of the shapes. Please read these lines carefully to familiarize yourself with screen coordinates and shape positioning.

Line 14 sets the text size to 12.

Lines 15, 16 implement the instructions that will be displayed in the destination window.

Lines 17, 32 implement the while loop in which the user will adjust the color of a patch.

Lines 18–22 draw in the backbuffer shapes and texts.

Line 23 flips from backbuffer to frontbuffer to display the shapes and texts.

Line 24 implements the `KeyCode` and `KeyIsDown` variables that will be used to manipulate the color of the right-hand square.

Lines 25–31 implement the conditional loop to change the color of the right-hand square. Note that the variable `adjustable` increases or decreases its value by 1 depending on `KeyCode`. You can use larger values than 1 for quicker changes. These lines are very important, since they are commonly used to manipulate shape characteristics. In this code we used `KeyCode`. As outlined above, this is a logical array containing all zeros except for the bit corresponding to the key that has been pressed. Each time the code runs to line 23, all `KeyCode` bits are set to zero until a `keypress` event occurs. In this code listing we have used the ASCII code corresponding to the left, right, and esc keys. If you do not want to remember these numbers, you can get the same code behavior using `KbName(KeyCode)`. For example, line 28 can be replaced with the following:

```
if strcmp(KbName(keyCode), 'esc')
```

Of course, this option takes more time, but if time is not an issue for your experiment, then use it to increase readability.

Line 33 writes to the backbuffer the last RGB value that has been assigned by the user.

Line 35 clears the buffer from any `keypress`. This trick is necessary because otherwise, the following `Kbwait` doesn't work, since the keyboard has been pressed to adjust the square color.

Line 36 waits for the user to press a key to display the string.

Line 37 shows the cursor again.

## *Placing Discs with the Mouse*

In this section we see how to place a disc on the screen and use the mouse to indicate where the disc is to appear. To do this, we will measure the Müller-Lyer illusion (Müller-Lyer 1889). It is one of the best-known geometric optical illusions consisting of two arrows, one with ends pointing in, and the other with ends pointing out. The next code is aimed at measuring the illusion magnitude in the arrows with ends pointing out, taking the participants' mouse button press.

**Listing 10.18**

```
try
   [w, rect] = Screen('Openwindow',0);
   cx = rect(3)/2;
   cy = rect(4)/2;
   myWidth = 2;
   mylength  = 200;
   arrowsize = 40;
   displacement = cx/2;
   myx = cx+displacement;
   Instructions= 'Click any mouse button to create a dot in correspondence
of the upper and lower limit of the line to your right.';
   Instructions2 = 'Click another time any mouse button when you are
satisfied with your match';
   Screen('TextSize', w,12) ;
   ShowCursor ('CrossHair')
   SetMouse (myx,cy)
   count=0;
   clicks=0;
   Usery=0;
   Userdata=zeros(2,1);
   while count<2
      Screen('DrawText',w, Instructions, 10, 20);
      Screen('DrawText',w, Instructions2, 10, 40);
      Screen('DrawLine',w,0,cx-displacement,cy+mylength,cx-displacement,cy-
mylength,myWidth);
      Screen('DrawLine',w,0,cx-displacement-arrowsize,cy-mylength-
arrowsize,cx-displacement,cy-mylength,myWidth);
      Screen('DrawLine',w,0,cx-displacement+arrowsize,cy-mylength-
arrowsize,cx-displacement,cy-mylength,myWidth);
      Screen('DrawLine',w,0,cx-displacement-
arrowsize,cy+mylength+arrowsize,cx-displacement,cy+mylength,myWidth);
     Screen('DrawLine',w,0,cx-displacement+arrowsize,cy+mylength+arrowsize,
cx-displacement,cy+mylength,myWidth);
     UserOval = [myx,Usery,myx+6,Usery+6];
     if clicks
        count=count+1;
        Screen('FillOval',w,0,UserOval);
        Userdata(count ,1)= Usery;
     end
     Screen('Flip',w );
     [clicks,Userx,Usery] = GetClicks;
   end
   Screen(w,'DrawText', sprintf('Your final length  was %d, press any key
to exit\n', Userdata(2,1)-Userdata(1,1)),10,60);
   Screen('Flip', w);
   while kbcheck end
      KbWait;
      ShowCursor;
      Screen('Close',w)
catch
   screen('Close',w)
   rethrow(lasterror)
end
```

### Analysis

Lines 3–4 use the `rect` argument to determine the screen-center pixel coordinates.

Lines 5–8 implement the variables for shaping the standard line. Please read these lines carefully to familiarize yourself with screen coordinates and shape positioning.

Line 9 implements the variable `myx` for mouse positioning.

Lines 10, 11 implement the instructions that will be displayed in the destination window.

Line 12 sets the text size to 12.

Line 13 sets the cursor to the cross-hairs shape.

Line 14 places the mouse in the right-hand side of the screen, in the middle y position.

Lines 15–18 implement variables to collect user button press.

Lines 19, 31 implement the while loop during which the user will click the mouse buttons.

Lines 20–26 draw in the backbuffer shapes and texts.

Lines 28–32 if the user has clicked the mouse, then draw the disc at the clicked position and save the y position in the `Userdata` variable to be displayed at the end of the experiment.

Line 33 flips from backbuffer to frontbuffer to display shapes and texts.

Line 34 gets user mouse click.

Line 36 writes in the backbuffer the length that has been assigned by the user.

Line 37 flips from backbuffer to frontbuffer.

Line 38 clears the buffer from any `keypress`. This trick is necessary because otherwise the following `Kbwait` doesn't work, since the keyboard has been pressed to adjust the square color.

## Summary

- PTB has many subsidiary functions that are useful in programming behavioral experiments.
- PTB makes it possible to get participants' responses from the keyboard.
- The keyboard response can be speeded up or not.
- PTB enables you to get the participants' responses from the mouse.
- Mouse and keyboard functions can also be used for letting the participant interact with the stimulus.

## Exercises

### *Exercise 1*

In Listing 10.18 we have programmed a code to measure the Müller-Lyer illusion in the condition in which the arrow's ends point out and it was presented to the left. Program a code in which the arrow's ends point in and it is presented to the right.

---

**Solution**

```
try
    [w, rect] = Screen('Openwindow',0);
    cx = rect(3)/2;
    cy=  rect(4)/2;
    myWidth = 2;
    mylength  = 200;
    arrowsize = 40;
    displacement = cx/2;
    myx = cx-displacement;
    Instructions= 'Click any mouse button to create a dot in correspondence
of the upper and lower limit of the line to your left.';
    Instructions2 = 'Click another time any mouse button when you are
satisfied with your match';
    Screen('TextSize', w,12) ;
    ShowCursor ('CrossHair')
    SetMouse (myx,cy)
    count=0;
    clicks=0;
    Usery=0;
    Userdata=zeros(2,1);
    while count<2
        Screen('DrawText',w, Instructions, 10, 20);
        Screen('DrawText',w, Instructions2, 10, 40);
        Screen('DrawLine',w,0,cx+displacement, cy+mylength,
cx+displacement, cy-mylength,myWidth);
        Screen('DrawLine',w,0,cx+displacement-arrowsize,cy-
mylength+arrowsize, cx+displacement,cy-mylength,myWidth);
        Screen('DrawLine',w,0,cx+displacement+arrowsize,cy-
mylength+arrowsize, cx+displacement,cy-mylength,myWidth);
        Screen('DrawLine',w,0,cx+displacement-arrowsize,cy+mylength-
arrowsize, cx+displacement,cy+mylength,myWidth);
        Screen('DrawLine',w,0,cx+displacement+arrowsize,cy+mylength-
arrowsize, cx+displacement,cy+mylength,myWidth);
        UserOval = [myx,Usery,myx+6,Usery+6];
        if clicks
            count=count+1;
            Screen('FillOval',w,0,UserOval);
            Userdata(count ,1)= Usery;
        end
        Screen('Flip',w );
        [clicks,Userx,Usery] = GetClicks;
    end
    Screen(w,'DrawText', sprintf('Your final length  was %d, press any key
to exit\n', Userdata(2,1)-Userdata(1,1)),10,60);
    Screen('Flip', w);
    while kbcheck end
    KbWait;
    Screen('Close',w)
catch
    Screen( 'Close',w)
    rethrow(lasterror)
end
```

---

**Analysis**

Lines 1, 9–12: are to catch any error after a Screen has been opened.

Line 2: implements the two arguments returned by the `Screen Openwindow` subfunction: `w` is the pointer to the window; `rect` is a vector containing the coordinates in pixels of the screen.

Line 3: implements the vector `col` having the three RGB values to get the red color.

Line 4: implements the variable `myWidth` that will be used to supply the line width to the `DrawLine` subfunction.

Lines 5–6: draw in the backbuffer a wide red line running diagonally across the screen.

Line 7: flips from backbuffer to frontbuffer to display the line.

Line 8: `Kbwhait` waits for user's input.

Line 9 closes the `w` window.

## A Brick for an Experiment

In our experiment, during each trial, the subject reports whether s/he perceived the discs as streaming or bouncing. Here in the brick this will be done using the function `KbWait` (because the response is not a reaction time). The first thing you need to do is to check how your response keys are coded. In the brick experiment the participant will press "`b`" if s/he sees the discs as bouncing and "`s`" if s/he sees the discs as streaming.

```
% set the keys we use in the experiment
bKey = KbName('b');
sKey = KbName('s');
```

Now we can get the subject's response with a while loop as we have seen previously in the chapter.

```
[secs, keyCode] = KbWait;
while keyCode(bKey)==0 && keyCode(sKey)==0
    [secs, keyCode] = KbWait;
end
```

Moreover, once the subject has pressed the key, we have to code the response. We could, for example, keep the response as is and have in the final data a long list of "r" and "s" values associated with each stimulus to which the subject has responded. However, it may be more convenient to code the subject's response as

"probability of bounce response" as in the original paper (Sekuler et al. 1997). If we decide to do this, we can encode the response as '1' (i.e., the probability of bounce response is '1') when the subject presses "b". As an alternative, and there is only one possible alternative, i.e., when the subject presses "s", we encode the response as '0' (i.e., the probability of bounce response is '0'). Therefore:

```
if keyCode(bKey) == 1
   pbounce = 1;
else
   pbounce = 0;
end
```

Now let's write everything into our SekulerExp function. Note that the rows of code we just wrote will be distributed in different places in the SekulerExp function. For example, the response keys variables sKey and bKey are declared at the beginning of the function. There is, in fact, no need to declare them during each trial before the response is collected. Note also that we have added the variable "response" to store the participant's response (i.e., pbounce) during the trials of the experiment.

**Listing 10.19**

```
function SekulerExp(InputDataStruct)
% M-script to realize an experiment based on crossmodal perception
% The experiment first performed by Sekuler, Sekuler, and Lau (1997)
% Author: Borgo, Soranzo, Grassi 2009
% EXPERIMENT`S SETTINGS
% get input data from the structure passed through the interface
nsub = InputDataStruct.nsub;
subname = InputDataStruct.subname;
subsex = InputDataStruct.subsex;
subage = InputDataStruct.subage;
nblock = InputDataStruct.nblock;
subnote = InputDataStruct.subnote;
isfixed = InputDataStruct.isfixed;
filename = InputDataStruct.filename;
% set the experiment details
conditions = [1, 1; 1, 2; 2, 1; 2, 2];
repetitions = 20;
if nsub == 0
    repetitions = 1;
end
% set the keys we use in the experiment
bKey = KbName('b');
sKey = KbName('s');
EventTable = GenerateEventTable(conditions, repetitions, isfixed);
TotalNumberOfTrials = length(EventTable(:, 1));
response = zeros(TotalNumberOfTrials, 1);
% opening operations for the screen function
try
    whichscreen = max(Screen('Screens'));
    [w, rect] =Screen('Openwindow', whichscreen, 0);
```

(continued)

**Listing 10.19** (continued)

```
31      refreshrate = Screen('FrameRate', w);
32      for trial = 1:TotalNumberOfTrials
33          % STIMULI (SELECTION)
34          VideoStimulusToPlay = EventTable(trial, 2);
35          SoundStimulusToPlay = EventTable(trial, 3);
36          % STIMULI (CREATION)
37          % STIMULI (PRESENTATION)
38          SoundToPlay = GenerateSound(SoundStimulusToPlay, w);
39          MakeVideoStimulus
40          % COLLECT SUBJECT'S ANSWER
41          [secs, keyCode] = KbWait;
42          while keyCode(bKey)==0 && keyCode(sKey)==0
43              [secs, keyCode] = KbWait;
44          end
45          if keyCode(bKey) == 1
46              pbounce = 1;
47          else
48              pbounce = 0;
49          end
50          response(trial) = pbounce;
51      end
52      % closing operation
53      Screen('CloseAll'); % close all
54      % STORE RESULTS
55  catch
56      Screen( 'Close',w)
57      rethrow(lasterror)
58  end
```

We now further modify the script to play the sound (rows 32–33, 42–43, 59) hide/show the cursor (rows34, 60) and save the data (rows 62–64):

**Listing 10.20**

```
1   function SekulerExp(InputDataStruct)
2   % M-script to realize an experiment based on crossmodal perception
3   % The experiment first performed by Sekuler, Sekuler, and Lau (1997)
4   % Author: Borgo, Soranzo, Grassi 2009
5   % EXPERIMENT`S SETTINGS
6   % get input data from the structure passed through the interface
7   nsub = InputDataStruct.nsub;
8   subname = InputDataStruct.subname;
9   subsex = InputDataStruct.subsex;
10  subage = InputDataStruct.subage;
11  nblock = InputDataStruct.nblock;
12  subnote = InputDataStruct.subnote;
13  isfixed = InputDataStruct.isfixed;
14  filename = InputDataStruct.filename;
15  % set the experiment details
16  conditions = [1, 1; 1, 2; 2, 1; 2, 2];
17  repetitions = 20;
18  if nsub == 0
19      repetitions = 1;
20  end
```

(continued)

**Listing 10.20** (continued)

```
% set the keys we use in the experiment
bKey = KbName('b');
sKey = KbName('s');
EventTable = GenerateEventTable(conditions, repetitions, isfixed);
TotalNumberOfTrials = length(EventTable(:, 1));
response = zeros(TotalNumberOfTrials, 1);
% opening operations for the screen function
try
    whichscreen = max(Screen('Screens'));
    [w, rect] =Screen('Openwindow', whichscreen, 0);
    refreshrate = Screen('FrameRate', w);
    InitializePsychSound;
    pahandle = PsychPortAudio('Open', [], [], 0, 44100, 1);
    HideCursor;
    for trial = 1:TotalNumberOfTrials
        % STIMULI (SELECTION)
        VideoStimulusToPlay = EventTable(trial, 2);
        SoundStimulusToPlay = EventTable(trial, 3);
        % STIMULI (CREATION)
        % STIMULI (PRESENTATION)
        SoundToPlay = GenerateSound(SoundStimulusToPlay, w);
        PsychPortAudio('FillBuffer', pahandle, SoundToPlay);
        PsychPortAudio('Start', pahandle);
        MakeVideoStimulus
        % COLLECT SUBJECT'S ANSWER
        [secs, keyCode] = KbWait;
        while keyCode(bKey)==0 && keyCode(sKey)==0
            [secs, keyCode] = KbWait;
        end
        if keyCode(bKey) == 1
            pbounce = 1;
        else
            pbounce = 0;
        end
        response(trial) = pbounce;
    end
    % closing operation
    Screen('CloseAll'); % close all
    PsychPortAudio('Close', pahandle);
    ShowCursor;
    % STORE RESULTS
    results = [EventTable, response];
    save(filename, 'results', 'nsub', ...
    'subname', 'subsex', 'subage', 'subnote')
catch
    Screen( 'Close',w)
    rethrow(lasterror)
end
```

## References

Kingdom F (1997) Simultaneous contrast: the legacies of Hering and Helmholtz. Perception 26(6):673–677

Müller-Lyer FC (1889) Optische Urteilstäuschungen. Archiv für Physiologie, Supplement Volume, 263–270

Sekuler R, Sekuler AB, Lau R (1997) Sound alters visual motion perception. Nature 385:308

## Suggested Readings

Tutorials on the Psychtoolbox can be found at the following web pages:

http://psychtoolbox.org/wikka.php?wakka=HomePage
http://psychtoolbox.org/wikka.php?wakka=PsychtoolboxTutorial