Chapter 9
Psychtoolbox: Video

The Psychophysics Toolbox (PTB) is a package for psychophysics research developed by David Brainard and Denis Pelli (Brainard 1997; Pelli 1997) and recently by Mario Kleiner (Kleiner et al. 2007). The PTB toolbox can be freely downloaded from the follwing website http://psychtoolbox.org/PsychtoolboxDownload. This toolbox has been used extensively over the last decade (the first version was released in 1995), and it is very useful for running experiments needing audiovisual stimuli. PTB routines treat the computer (Linux, Mac, or Windows) as a display device, i.e., a frame buffer, a portion of memory generally placed within the graphics card where images are temporally stored. To do this, PTB interfaces MATLAB with a low-level computer language such as C. Hence, besides a number of .m functions, PTB includes low-level information included in MEX files. The most important MEX file is Screen.mex, which will be described in the current chapter.

PTB includes a great number of functions, whose documentation is displayed at http://docs.psychtoolbox.org/Psychtoolbox.

The Screen Function

As you may have anticipated, the core of PTB is the Screen function. It includes a number of subfunctions allowing for accurate control of the images presented on the computer screen. To see all the subfunctions of Screen (and a partial help), type "`Screen`" with no arguments at the MATLAB prompt (if you type "`help Screen`" you get a general introduction about the function). Each time you call the Screen function it outputs, at the MATLAB prompt, information about your graphics hardware. Before running any experiment it is important to read this information to make sure you are equipped for your needs. However, you may not want to have

M. Borgo et al., *MATLAB for Psychologists*,
DOI 10.1007/978-1-4614-2197-9_9, © Springer Science+Business Media, LLC 2012

this information repeatedly prompted each time the Screen function is called. In this case, you can use the following code[1] to suppress it:

```
Screen('Preference','SuppressAllWarnings',1);
```

The general call of the Screen function is the following:

```
[v1, v2, …]=Screen('Sub-functionName', parameter1, parameter2, …)
```

The Screen function always needs the sub-function name together with the parameters of the sub-function. Moreover, the function returns a number of variables (from zero to many) according to the specific subfunction. The help of its subfunctions can be seen by typing `Screen('Sub-functionName?')` at the MATLAB prompt. For example:

```
Screen('OpenWindow?')
Screen('FillRect?')
```

Or equivalently

```
Screen OpenWindow?
Screen FillRect?
```

In the displayed help, optional input arguments are preceded by a comma, whereas nonoptional arguments are not. For example:

```
[VBLTimestamp StimulusOnsetTime FlipTimestamp Missed Beampos] = Screen('Flip',
windowPtr [, when] [, dontclear] [, dontsync] [, multiflip]);
```

Here `when`, `dontclear`, `dontsync` and, `multiflip` are optional arguments (i.e., they have default values), whereas `windowPtr` is not. Since the order of the function arguments cannot be modified, empty square brackets can be used to reach the desired position. Hence, in the above example, to change the `dontsync` default value, we need to write:

```
Screen('Flip', windowPtr, [], [], dontsync)
```

The following is an unsorted list of the things that can be done with `Screen`, which are usually done by a specific subfunction. `Screen` can be used to get information about the screen such as the refresh rate or the size in pixels. `Screen` can also be used to show strings (very useful in psycholinguistics experiments), to draw shapes such as lines, ovals, rectangles, or any other kind of geometrical shape you may want to draw. `Screen` can also import pictures from graphics files (such as .jpg, or .tif) saved on the hard drive and can be used to create video clips. But probably the most interesting feature of `Screen` is that everything is done with maximal timing accuracy. This is because the stimulus presentation is synchronized with the monitor refresh rate.

In the current chapter we first show how to use Screen to get information about the hardware and software characteristics. Then we show how to draw figures and

[1] Please note that the codes presented in this book works with Psychtoolbox from version 3.

text, and how to import pictures from external files. Finally, we show how to create a sequence of events and how to present them with great timing accuracy.

As anticipated, the `Screen` function, the main function of PTB, is written in a low-level computer language. Because of this, when you use `Screen`, if your code crashes it may be difficult to go back into your script or to the MATLAB prompt (for example, you might need to quit MATLAB from the task manager or, even worse, to switch off your computer!). To avoid this problem, we recommend writing all code within a `try` and `catch` block (see Chap. 3). This trick bypasses some of the problems that may arise if your program crashes. The following example shows how you should use the try–catch commands.

Listing 9.1

```
1 try
2    % write here the code written in the chapter
3 catch
4    Screen('CloseAll')
5    rethrow(lasterror)
6 end
```

Analysis

Lines: 1, 3–6 are to catch any error after a Screen has been opened.

Line 2: the script written in the chapter.

Lines 4–5: closes all screens in case the program written in the try section crashes. Moreover, it reports the last error found in the try section. This is useful for debugging the code.

Another option is to use an auxiliary monitor so to keep the MATLAB command window on one monitor and display the figures created with PTB on the other one. In this way, if the code crashes, it would be possible reading at the MATLAB prompt the error generating the crash and, in most of the cases, it would be possible to "close" the screen code.

How to Use `Screen` to Get Information

The function `Screen` can be used to get information about the PTB itself as well as the characteristics of the computer in use. In particular, the Screen subfunctions are useful for increasing a program's portability. For example, if you are displaying a video clip, it is important to get the screen refresh rate of the computer in use to produce the same visual effect in terms of timing when different machines are used. The following table explains some of these subfunctions.

Sub function	Command	Example
Version	`struct=Screen('Version');`	Return a structure with the characteristics of the PTB
Computer	`comp=Screen('Computer');`	Return a structure with the characteristics of the computer
Screen	`screens=Screen('Screens');`	Return an array of numbers (0, 1, 2, …). Each number identifies one screen connected to the computer. The default, 0, is the screen with the menu bar
Rect	`rect=Screen('Rect', screenNumber);`	Return an array with the top left corner (always 0, 0) and the bottom right corner (n, m) coordinates of the screen. The number of the bottom right corner coincides with the screen's resolution (e.g., 1,024×768). ScreenNumber is a pointer (i.e., the one returned by the Screens subfunction) that tells the function which screen-rect is to be returned
FrameRate	`hz=Screen('FrameRate', screenNumber);`	Returns the refresh rate of the screen identified by the pointer screenNumber
GetFlipinterval	`[monitorFlipInterval nrValidSamples stddev]=Screen('GetFlipInterval', windowPtr [, nrSamples] [, stddev] [, timeout]);`	Returns the flip interval (in seconds), i.e., the interval in seconds between two consecutive vertical retraces of the screen identified by the pointer. This subfunction has to be run after the OpenWindow subfunction

How to Use `Screen` to Draw Figures

Preliminary Notions: Drawing Figures in Three Steps—Opening, Drawing, and Closing

The main thing `Screen` is used for is to draw figures and to present them with maximal timing accuracy. Generally speaking, there are three main figure types: figures drawn with PTB, imported figures (e.g., .jpg, .tif, …) and text figures. Independently from the type of figure you are drawing, the drawing is done in three steps that we can call opening, drawing, and closing. These steps are normally found in any program that displays figures.

Opening the Window

The "opening" step is controlled by the subfunction `'OpenWindow'`:

```
[MyScreen, rect] = screen('OpenWindow', 0, [0, 255, 0]);
```

When the opening is done, we take control of the screen where we are going to draw the figure. The first argument of the `'OpenWindow'` subfunction indicates the screen we want to refer to. Indeed, many screens can be connected to the same computer at the same time. The screen with the menu bar is identified with the default number "0".

Therefore, with the command line above, we get control of the screen with the menu bar. To this particular screen we also assign a name (i.e., the returned pointer `MyScreen`). Therefore, later in the code, every time we need to access this screen, we use screen's name (`MyScreen`) rather than 0. The above code paints the whole screen in green color using the RGB triplet `[0, 255, 0]`. In the `Screen` function, the color argument is passed either with a single gray value (i.e., 0–255) or with a RGB triplet. By default, if the color argument is omitted, the screen is white, i.e., the default value for this argument is `[255 255 255]`. Last but not least, '`OpenWindow`' returns the screen size, which above we saved in the `rect` variable. The `rect` variable is an array of the screen coordinates in pixels. The first two coordinates are those of the top left corner; the second two coordinates are those of the bottom right corner. The top left coordinates are `0,0`, whereas those of the bottom right depend on screen resolution.

The full list of arguments for the `OpenWindow` subfunction is the following:

```
Screen('OpenWindow',0[,color][,rect][,pixelSize][,numberOfBuffers]…
                     [,stereomode][,multisample][,imagingmode]);
```

The following table outlines the use of these arguments.

Opt. argument	`Color`
Description	Can be a scalar, if you want an achromatic screen or a 1×3 vector with the unnormalized RGB values (i.e. 1–256). For an RGB color description, refer to Chap. 5
Example	`Screen('OpenWindow', 0, [255,0,0])` `% it turns the screen red`

Opt. argument	`Rect`
Description	Specifies the size, in pixels, of your window (this option works better under Mac OS). This is a 1×4 vector with the coordinates of the window. The first two numbers refer to the x and y coordinates (in pixels) of the upper left corner, respectively; the last two numbers refer to the x and y coordinates of the bottom right corner of the window, respectively. Hence, there is the possibility to draw the stimuli even in a portion of your screen instead of the whole screen, although you'll probably never use it. The following example turns the screen red using the specified coordinates
Example	`Screen('OpenWindow', 0, [255, 0, 0],` `[250,250,450,650])`

Opt. argument	`pixelSize`
Description	Specifies the number of bits per pixel devoted to creating colors in the screen. Usual numbers of bits are 8, 16, or 32 per pixel, corresponding to 256, thousands, or millions of colors, respectively. This depends on the graphics card. If you do not know how many bits per pixel your graphics card can use, you can ask PTB using the Screen subfunction PixelSizes. By default, PTB works with 8 bits, and therefore colors are coded within the 0–255 (i.e., 2^8) range. If a different number of bits is used, the range defining the color also changes. For example, with 16 bits, the triplet for the red color is [65535, 0, 0]
Example	`Screen('Openwindow',0,[],[],16)`

Opt. argument	`numberOfBuffers`
Description	Determines the number of buffers to use. You normally use one or two buffers (see next chapter) to run your experiments, so you don't change this parameter except for testing or debugging reasons
Example	`Screen('Openwindow',0,[],[],[],2)`

(continued)

(continued)

Opt. argument	`Stereomode`
Description	PTB offers different stereo possibilities. The default value of this argument is 0, that is, Monoscopic viewing. If you are equipped with stereo hardware, such as shutter glasses, you can opt for values from 1 to 3 of stereomode according to where you want the images for the left and right eyes appearing on the screen If you want a stereogram, then set stereomode to 4. In this way you split the screen into two halves, where the left view is for the left eye; set stereomode to 5 for cross-fusion. If you are equipped with color glasses for anaglyph stereo vision, then set stereomode to 6, 7, 8, or 9 for different combinations of colors Finally, set stereomode to 10 if you have two monitors: you will have one image per monitor
Example	`Screen('Openwindow',0,[],[],[],[],4)`

Opt. argument	`"multisample"`
Description	This parameter enables an antialiasing procedure. In brief, when you design your stimuli for a screen, aliasing is a problem that occurs when the approximation due to pixel size is not good enough. When you draw a disc, for example, its smoothness can be very poor if the resolution of the screen in not high enough. Or, when you create video clips, you may have a temporal aliasing problem resulting from the limited frame rate. When multisample is greater than 0, PTB looks for the best antialiasing solution that can be obtained by your hardware. This will improve the quality of the stimuli; however, the downside in doing this is the increase of video memory use, leading to loss of precision in terms of time. You might not want this if you are collecting reaction times
Example	`Screen('Openwindow',0,[],[],[],[],[],5)`

Opt. argument	`Imagingmode`
Description	This parameter enables PTB's internal image-processing pipeline. The pipeline is off by default. By setting this parameter to 1, you enable this feature to perform image-processing operations that are executed on the graphics processor itself
Example	`Screen('Openwindow',0,[],[],[],[],[],[],0)`

Drawing: An Introduction

This is the step where we actually design the figures. We'll see later in the chapter how to program some figures that may be drawn directly within the PTB environment. But first, we have to introduce another subfunction. Indeed, in order for a drawing to be effective, it has to be followed by the subfunction `Flip`.

The figure is automatically drawn in the background memory (also called backbuffer), which is not visible. The `Flip` subfunction moves the previously drawn figure from the backbuffer to the foreground memory (the frontbuffer) so that it becomes visible on screen. This "flip" of the figure from the backbuffer to the frontbuffer

occurs at a specific time, which is the first available vertical retrace of the screen. The flip action is performed on any figure found in the background memory. In other words, if you have drawn, let's say, a square, a circle, and a triangle and you want to show them on the screen, you do not need to flip each object, but instead, you can flip all objects at once. The `Flip` subfunction needs to be addressed to a particular screen (i.e., to a particular screen pointer). In everyday situations, the screen we address to is the only screen connected to the computer. However, if you are using two or more screens, you could decide on which screen the figure has to be flipped. When the `Flip` subfunction is executed, the background memory is cleared. Therefore, if you have nothing in the background and you call the `Flip` subfunction, everything that is currently in your foreground is deleted, because it is replaced with an empty object.

The complete Flip command is the following:

```
[VBLTimestamp StimulusOnsetTime FlipTimestamp Missed Beampos] = Screen('Flip',
 windowPtr [, when] [, dontclear] [, dontsync] [, multiflip]);
```

We will see later how to use some of the Flip options to control the timing of the stimuli.

Closing

Closing ends the code, and in the majority of cases may look simply like the following command:

```
screen('CloseAll');
```

The `CloseAll` subfunction closes all the figures created during code execution and sets the screen back to normality, returning control to MATLAB. It is important to close everything, otherwise, the control is not returned to MATLAB. In PTB, you can close all objects at once (for example, with the subfunction CloseAll) or close only specific objects (remember, however, that at the end of your code you need to close all the objects created). This can be done as follows:

```
screen('Close', objectPointer)
```

where objectPointer is the pointer to the object you want to close. The possibility of closing selected objects is particularly interesting if the code you are writing needs a lot of memory. Overall, the amount of memory you are using depends on the number of objects and on their size. If memory consumption becomes critical for your program, then it might be helpful closing the objects no longer in use.

Drawing: Reprise

Now that the preliminaries are out of the way, we can have a go at doing some drawing. Replace the text comment in the try–catch example (presented in Listing 9.1) as in Listing 9.2.

Listing 9.2

```
1  try
2     screens = Screen('Screens');
3     whichscreen = max(screens);
4     [myscreen,rect]=Screen('OpenWindow',whichscreen, [0, 255,0]);
5     KbWait;
6     screen('CloseAll');
7  catch
8     screen('CloseAll')
9     rethrow(lasterror)
10 end
```

Analysis

This short script does a few things:

On line 2: It looks for how many screens are connected to the computer.

On line 3: if there is more than one screen, it selects one of them without the menu bar and assigns an arbitrary name to it. Later, we'll open this screen and turn it green.

On line 4: the screen assumes the name `myscreen` (i.e., its pointer), and we save the coordinates of the screen in the variable `rect`. Moreover, we paint the full screen `myscreen` green.

On line 5: we make the program wait for a key-press event. This is done by means of the `KbWait` command, which stops the execution of the code until the user presses a keyboard key.

On line 6: The subfunction "`CloseAll`" returns control to MATLAB.
In the previous example, we did not draw any figures on the screen. In the next section we will extend the code by drawing some objects.

Drawing Shapes

To understand how to draw geometric shapes on the screen, we extend Listing 9.2 to make the stimulus for a color afterimage. To do this, we want to draw a red square (400 by 400 pixels) on a green background, and to do this we use the subfunction `FillRect`, which draws a filled rectangle; use the subfunction `FrameRect` to get a framed rectangle. Both the `FillRect` and `FrameRect` subfunctions take as parameters the screen pointer followed by the color (in RGB triplet form) and finally the coordinates of the rectangle we want to draw. When the `FillRect` or `FrameRect` subfunctions are called, a rectangle is drawn in the backbuffer. To actually see the rectangle on the screen, we need to call the Flip subfunction.

Listing 9.3

```
try
   screens = Screen('Screens');
   whichscreen = max(screens);
   [myscreen, rect] = Screen('OpenWindow', whichscreen, [0, 255, 0]);
   square_size = [0, 0, 400, 400];
   square_rect = CenterRect(square_size, rect);
   Screen('FillRect', myscreen, [255, 0, 0], square_size);
   Screen('Flip', myscreen);
   KbWait;
   Screen('CloseAll');
catch
   screen('CloseAll')
   rethrow(lasterror)
end
```

Now if you look at the red square long enough (e.g., 20 s) and then move your eyes to a white sheet of paper, you can see the afterimage of the red color.

You may have noticed in the script above that we used the very handy function `CenterRect`. It calculates the coordinates to include one figure within the center of another one whose coordinates are known. In the example, the external figure is the entire screen; therefore, the function centers the figure within the screen.

`CenterRect` and the other functions for manipulating coordinates (type help `PsychRets` at the MATLAB prompt or see below) can also be used in a nested way. Let us now display a simultaneous lightness contrast display. We divide the screen into two sectors, one white and one black, and at the center of each sector we need to place a gray square. Everything is done in the following example.

Listing 9.4

```
try
   screens = Screen('Screens'); % count the screens
   whichscreen = max(screens); % select the screen
   [myscreen, rect] = Screen('OpenWindow', whichscreen,
   % create the black/white sectors
   sector = [0, 0, rect(3)/2, rect(4)]; % set the size o
   sector_l = sector;
   sector_r = AdjoinRect(sector, sector_l, RectRight);
   Screen('FillRect', myscreen, 255, sector_l);
   Screen('FillRect', myscreen, 0, sector_r);
   % create the left/right grey square
   square = [0, 0, 100, 100];
   square_l = CenterRect(square, sector_l);
   square_r = CenterRect(square, sector_r);
   Screen('FillRect', myscreen, 255/2, square_l);
   Screen('FillRect', myscreen, 255/2, square_r)
   Screen('Flip', myscreen);
   KbWait;
   Screen('CloseAll');
catch
   screen('CloseAll')
   rethrow(lasterror)
end
```

The script is very similar to Listing 9.3. In the example, we also introduced the `AdjoinRect` function which adds a rectangle at the right of the specified `sector`. Note also that here we used `CenterRect` in a nested way, because now "center" is the center of the defined sector, not the center of the screen. When you are drawing the stimuli for your experiment, our suggestion is to proceed as follows. First, set the size of the shape, regardless of its position on the screen, by setting its size starting from the top-left corner. Then use the `PsychRect` function to define the actual coordinates on the screen.

In the following table can be found a complete list of the functions that can be used to simplify your work that uses the rect coordinates. Please refer to the Psychtoolbox help for more complete information on how to use them.

Function	Description
AdjoinRect	Moves a rect next to another one
AlignRect	Aligns a rect over another one
ArrangeRects	Arranges an array of rects in a pleasant way
CenterRect	Centers a rect within a second one
CenterRectOnPoint	Centers a rect around given x,y coordinates
CenterRectOnPointd	Centers rect around an x,y coordinate pair
ClipRect	Returns the intersection of two rects
ClipRect	Returns the intersection of two rects
InsetRect	Shrinks/expands rect by additive insets
IsEmptyRect	Returns 1 if empty, returns 0 otherwise
IsInRect	Is the point inside a rect?
OffsetRect	Shifts rect vertically and horizontally
RectBottom	Returns index of yBottom entry of a rect
RectCenter	Returns the integer x,y coordinates of center
RectCenterd	Returns the exact x,y coordinates of center
RectOfMatrix	Accept an image as a matrix and returns a PTB rect specifying the bounds
RectHeight	Returns the height of a rect
RectLeft	Returns index of xLeft entry of a rect
RectRight	Returns index of xRight entry of a rect
RectTop	Returns index of yTop entry of a rect
RectWidth	Returns width of a rect
RectSize	Returns the width and the height of a rect
ScaleRect	Scales a rect by multiplicative factors
SetRect	Creates a rect (i.e., a vector) from four input coordinates
SizeOfRect	Accepts a Psychtoolbox rect [left, top, right, bottom] and returns the size [rows columns] of a MATLAB array (i.e. image) just big enough to hold all the pixels
UnionRect	Smallest rect containing two given rects

To draw filled ovals (including circles) we use the subfunction `FillOvals` instead of the subfunction `FillRects`. The counterpart of `FrameRect` is `FrameOvals`. In the following table, we present a list of the shapes that can be drawn with PTB. More-complex graphical shapes can be drawn by combining two or more figures.

Sub/Function	Command	Description
DrawLine	Screen('DrawLine', windowPtr [,color], fromH, fromV, toH, toV [,penWidth]);	draws a line
DrawArc	Screen('DrawArc',windowPtr,[color],[rect],startAngle,arcAngle)	draws a circular arc unfilled with color (i.e., a Pac-Man-like figure)
FrameArc	Screen('FrameArc',windowPtr,[color],[rect],startAngle,arcAngle[,penWidth] [,penHeight] [,penMode])	as above
FillArc	Screen('FillArc',windowPtr,[color],[rect],startAngle,arcAngle)	as above but filled with color
FillRect	Screen('FillRect', windowPtr [,color] [,rect]);	draws a rectangle filled with color
FrameRect	Screen('FrameRect', windowPtr [,color] [,rect] [,penWidth]);	draws a rectangle unfilled with color
FillOval	Screen('FillOval', windowPtr [,color] [,rect] [,perfectUpToMaxDiameter]);	draws a filled oval
FrameOval	Screen('FrameOval', windowPtr [,color] [,rect] [,penWidth] [,penHeight] [,penMode]);	draws a framed oval
FramePoly	Screen('FramePoly', windowPtr [,color], pointList [,penWidth]);	draws a framed polygon
FillPoly	Screen('FillPoly', windowPtr [,color], pointList [, isConvex]);	draws a filled polygon

Batch Processing: Drawing Multiple Figures at Once

It is often useful to be able to draw multiple figures at once. This operation is not only useful, but it is also an efficient operation to do with PTB 3. The repeated drawing is achieved using the functions that we have used so far. Instead of writing repeatedly the same drawing subfunctions:

```
Screen('FillRect', win, [red1 green1 blue1], [left1 top1 right1 bot1]);
Screen('FillRect', win, [red2 green2 blue2], [left2 top2 right2 bot2]);
…
Screen('FillRect', win, [redn greenn bluen], [leftn topn rightn
botn]);
```

you can write in the following:

```
mycolors = [red1, red2,… ; green1, green2,… ; blue1, blue2,…];
myrects = [xtop_1, xtop_2,…; ytop_1, ytop_2,…; xbottom_1, xbottom_2,…;
ybottom_1, ybottom_2,…];
Screen('FillRect', win, mycolors, myrects);
```

In other words, you first write a 3×n matrix for the RGB values, and then a 4×n matrix for the figures' coordinates. These matrices are then passed to the desired subfunction to draw the shapes. In the 3×n color matrix, each row identifies the RGB values for the corresponding figure. In the same way, in the 4×n matrix for the figures' coordinates, each column identifies the coordinates of the corresponding figure; odd rows are the `x` coordinates, and even rows are the `y` coordinates.

In the following example, we draw a setting that could be used for a Posner-like experiment by drawing three frames placed one after the other in the middle of the screen. The left and the right frames are gray, while the middle frame is black so that it cannot be seen. Note that colors and coordinates are transposed when they are passed to the `FrameRect` subfunction.

Listing 9.5

```
try
   whichscreen = max(Screen('Screens')); % count the screens
   [w, rect] = Screen('OpenWindow', whichscreen, 0); % open it
   midsquare = CenterRect([0, 0, 100, 100], rect);
   squares = [AdjoinRect(midsquare, midsquare, RectLeft); ...
   midsquare; ...
   AdjoinRect(midsquare, midsquare, RectRight)];
   colors = [255/2, 255/2, 255/2; ...
   0, 0, 0; ...
   255/2, 255/2, 255/2];
   Screen('FrameRect', w, colors', squares', 5);
   Screen('Flip', w);
   KbWait;
   Screen('CloseAll');
catch
   screen('CloseAll')
   rethrow(lasterror)
end
```

Drawing Text

'DrawText' is the subfunction to draw text on the screen. The subfunction's options are the following:

```
[newX,newY]=Screen('DrawText', windowPtr, text [,x] [,y] [,color]
[,backgroundColor] [,yPositionIsBaseline]);
```

where windowPtr is the pointer to the screen, and text is the string of text you want to draw. Optional arguments are the x and y coordinates where the text starts (these coordinates refer to the top left corner) and the text color. Note that in this subfunction the color argument is passed after the coordinates, instead of before, as was the case for rectangles and ovals. Further optional arguments are backgroundColor and yPositionIsBaseline. backgroundColor is the color behind the text (It does not seem to work properly under Windows and Linux. However, this problem can be easily solved by drawing a colored rectangle before drawing the text.) yPositionIsBaseline is a logical value; if true, the y coordinate for the text refers to the bottom, instead of the upper, part of the text.

The Drawtext subfunction returns the x and y coordinates of the end of the text. This is useful because the strings you are writing could be of a different lengths, covering a different number of pixels. Therefore, knowledge of the ending coordinates of a string is important for arranging two or more strings of text. If you simply need to write one string in the middle of the screen, you can use the function DrawFormattedText:

```
[nx, ny, textbounds] = DrawFormattedText(win, tstring [, sx][, sy][, color]
[, wrapat][, flipHorizontal][, flipVertical][, vSpacing])
```

DrawText accepts not only coordinates expressed in pixels but also the option 'center', which can be used to center the text on the screen either on the horizontal or vertical axis. The following code writes a string starting from the middle of the screen.

Listing 9.6

```
1  try
2     screens = Screen('Screens');
3     whichscreen = max(screens)
4     [w, rect] =Screen('Openwindow', whichscreen);
5     mytext = 'MATLAB';
6     DrawFormattedText(w, mytext, 'center', 'center');
7     Screen ('Flip', w);
8     KbWait;
9     Screen('Close', w)
10 catch
11    Screen('Close',w)
12    rethrow(lasterror)
13 end
```

PTB provides additional writing subfunctions specifying the style, font, mode, and size of the text. To get these features the corresponding subfunctions have to be called before `DrawText` or `DrawFormattedText` is called. These subfunctions allow for *getting* and *setting* text features at the same time; that is, the same function *gets* the type of text that is currently on and *sets* the desired text type. Let us see how they work.

`'TextStyle'` specifies the text style. 0 is normal, 1 is bold, 2 is italic, 3 is bold and italic, 4 is underline, 5 is bold and underline, 6 is italic and underline, and 7 is italic, bold, and underline.

For example, the command:

```
previous_style = Screen('TextStyle', w, 2);
```

returns the style that was previously in use (0 is the default style) and sets the style to bold for next text.

`'TextFont'` specifies the text font; it can be invoked by passing the font name or via the font number. The subfunction returns two arguments, which are the number and name of the font that is currently in use (this is because to each font there also corresponds a number). Use the following syntax:

```
[previousFontName, previousFontNumber] = Screen ('TextFont',w, 'Verdana');
```

The function returns both previous name and number style and sets Verdana as the style for future text.

`'TextMode'` specifies the text mode; there are 16 different modes, ranging from normal to dashed, dot-dashed, and so on. It works only with Mac OS. Use the following syntax to get the previous mode and to set the new one:

```
previous_mode = Screen('TextMode', w,10);
```

`'TextSize'` specifies the text size. Use the following syntax to get the previous size and to set the new one:

```
previous_size=Screen('TextSize', w,40) ;
```

Finally, there is the '`TextWidth`' subfunction, which returns the horizontal offset, that is, the change in the horizontal pen position that will be produced by the string. That is, if you are not sure how many bytes your string is, use '`TextWidth`' to get it. Use the following syntax:

```
Width=Screen('TextWidth', w, mystring);
```

where `mystring` is the string that you are to type. It returns a negative number of bytes if you write from right to left. `Mystring` may include 2-byte characters (e.g., Chinese).

The following code listing shows how to use some of these subfunctions. Moreover, it provides an example of changing text. The word MATLAB is written three times in three different colors that are continuously changing. The word is also vertically and horizontally flipped using some of the options of `DrawFormattedText`.

Listing 9.7

```
try
   screens = Screen('Screens');
   whichscreen = max(screens);
   [w, rect] =Screen('Openwindow', whichscreen, 0);
   mytext = 'MATLAB';
   Screen('TextSize',w, 50);
   Screen('TextFont',w, 'Arial');
   for r = [0:2:255, 254:-2:0]
       DrawFormattedText(w, mytext, 'center', 'center', [r, 0, 0]);
       Screen('Flip', w);
   end
   for g = [0:2:255, 254:-2:0]
       DrawFormattedText(w, mytext, 'center', 'center', [0, g, 0], ...
       [], 1, 0);
       Screen('Flip', w);
   end
   for b = [0:2:255, 254:-2:0]
       DrawFormattedText(w, mytext, 'center', 'center', [0, 0, b], ...
       [], 0, 1);
       Screen('Flip', w);
   end
   Screen('Close', w)
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Importing Images

`Screen` can be used to import and to show images stored on the hard drive. Images are shown in three steps. First you need to load the image in the MATLAB workspace (see Chap. 5); then, you need to create a *texture* of the picture, and finally, you can show the texture on the screen. Let us analyze these steps by running the code in Listing 9.8.

Listing 9.8

```
1  try
2     load mandrill
3     [row, col] = size(X);
4     r = [0, 0, row, col];
5     [windowPtr, rect] = Screen('OpenWindow', 0);
6     r = CenterRect(r, rect);
7     X2=ind2rgb(X,map);
8     X2=X2*256;
9     pic = Screen('MakeTexture', windowPtr, X);
10    Screen('DrawTexture', windowPtr, pic, [], r, 45);
11    Screen('Flip', windowPtr);
12    KbWait;
13    Screen('CloseAll');
14 catch
15     Screen('Close',w)
16     rethrow(lasterror)
17 end
```

Analysis

On lines 2 and 3 we load the image, get its size, and store it in the MATLAB workspace.

On line 4 we set a rectangle as large as the picture, and change its coordinates so that it is set at the center of the screen.

On line 7 we change the indexed image format into an RGB format using the MATLAB function `ind2rgb`. We need to do this because PTB works with intensity matrices (i.e., gray-scale or RGB) instead of indexed matrices.

On line 8 we multiply the resulting RGB matrix by 256. This is because PTB expects integers ranging from 0 to 255, while the values returned by `ind2rgb` are in the 0–1 range.

On line 9 we transform the picture into a PTB texture. PTB uses OpenGL[2] commands, and a texture can be seen as a sort of image in OpenGL. A discussion of OpenGL technology is beyond the scope of the present text. However, it is important to know that we need to 'remap' every image into a *texture* element.

On line 10 we draw the texture image in the background memory.

On line 11 we show the picture on the screen.

It has to be stressed that the matrix of an image cannot be directly displayed: the matrix has to be converted into a texture before being displayed.

[2] OpenGL's main purpose is to render two- and three-dimensional figures into a frame buffer. These figures are described as sequences of vertices (which define geometric objects) or pixels (which define images). OpenGL performs several processing steps on these data to convert them into pixels to create the image in the frame buffer.

Video Clips

PTB functions can be used to create video clips such as a figure moving along the screen, where a video clip is a succession of static pictures. Each picture is called frame. The possibility of drawing video depends on the refresh rate and on the pixel size. These two factors affect the granularity of the motion. A displacement cannot be lower than the size of one pixel. Similarly, frames cannot be presented at a faster rate than the refresh rate.

In the majority of cases, video clips are drawn using for loops, as in the following example. Here, a black disc on a white background moves horizontally from left to right on the screen.

Listing 9.9

```
1  try
2     screens = Screen('Screens');
3     whichscreen = max(screens);
4     [w, rect] =Screen('Openwindow', whichscreen, 0);
5     discdiam = 20;
6     disc = [0, 0, discdiam, discdiam];
7     rectinrect = [0, 0, rect(3)/2, rect(4)/2];
8     rectinrect = CenterRect(rectinrect, rect);
9     disc = AlignRect(disc, rectinrect, 'center', 'left');
10    for i = 0:600
11       Screen('FillRect', w, 255, rectinrect);
12       Screen('FillOval', w, 0, [disc(1)+i, disc(2), disc(3)+i, disc(4)]);
13       Screen('Flip', w);
14    end
15    Screen('CloseAll'); % close all
16 catch
17     Screen('Close',w)
18     rethrow(lasterror)
19 end
```

Analysis

On line 6 we implement the variable `disc`, which is a quadruplet of coordinates for the disc; on lines 7 and 8 we implement `rectinrect`, which is a quadruplet of the coordinates for the frame within which the disc will move.

From line 10 to line 14 we implement the "for loop" in which the x coordinates of the disc change. At the first iteration (i=0) the disc is placed at the left border of the frame. When `i` equals `1`, the x coordinate of the disc is augmented of one unit; therefore the disc is drawn 1 pixel further to the right than the previous disc. When `i` equals `2`, the shift of the discs becomes 2 pixels, and so on up to `i = 600`, when the discs disappear behind the frame.

In practice, every time you iterate a loop you need to redraw the object you want to move in a different position by specifying the new x and y coordinates where you want the figure to be drawn. Of course you can set into motion any figure you want. Moreover, motion can be done not only along a straight line but

also with a certain fuzziness. Here the string "hello" moves from left to right in an "uncertain" way.

Listing 9.10

```
1  try
2     screens = Screen('Screens');
3     whichscreen = max(screens);
4     [w, rect] =Screen('Openwindow', whichscreen, 0);
5     frame = CenterRect([0, 0, rect(3)/2, rect(4)/2], rect);
6     x = frame(1);
7     y = rect(4)/2;
8     Screen('TextSize', w, 24);
9     for i = 0:600
10       Screen('FillRect', w, 255, frame);
11       Screen('DrawText', w, 'hello world!', x, y);
12       Screen('Flip', w);
13       x = x + (round(rand)*4)-1;
14       y = y + (round(rand)*2)-1;
15    end
16    Screen('CloseAll'); % close all
17 catch
18     Screen('Close',w)
19     rethrow(lasterror)
20 end
```

Analysis

In the current example, the x and y coordinates of the string "hello world!" are modulated by a random factor (using the function *rand* on lines 13 and 14). On line 14, y is made to change by either −1 (a displacement to the upper part of the screen), 0 (no displacement), or +1 (a displacement toward the bottom part of the screen). Therefore, on average, the figure oscillates along the y axis without progressing in any particular direction. However, on line 13, the x coordinate can change by −1, 0, +1, +2, or +3. Negative changes move the string toward the left, whereas positive changes move the string toward the right. Because the randomly generated number is more likely to be positive than negative, the string eventually moves toward the right of the screen.

Drawing Things at the Right Time

Up to now, we have not paid yet much attention to timing. However, on many occasions, timing is an important issue for our stimuli, because we need to be able to control their duration.

Timing is intrinsically connected to the screen refresh rate. For example, you cannot present objects whose duration lasts less than one refresh per cycle. By the same token, you cannot present a stimulus for a duration that is not an exact

multiple of the screen refresh rate. The reason is the following. Let's suppose you are showing one stimulus (s1) and that this stimulus has to be replaced by the next stimulus (s2). If the duration of s1 on screen is not a multiple of the refresh rate, it may happen that the drawing of s2 begins when s1 is still on screen. Therefore, when you are deeply concerned about timing, always use durations that are multiples of the screen refresh rate.

Independently from the stimulus type, timing is controlled by two optional arguments of the subfunction `Flip`. These arguments are "`when`" and "`VBLTimestamp`". "`when`" tells the flip subfunction when the flip from foreground to background is to be done. "`VBLTimestamp`" is the time when the flip has actually been done. Both "`when`" and "`VBLTimestamp`" are expressed in seconds and refer to the system time, a timer that is switched on when you switch on your computer.

The following example illustrates how to control the timing. We first generate a fixation point which stays on the screen for 0.5 seconds and then a red square appears. After 0.75 seconds the red square is cleared, and after 1 seconds a green square appears for 0.6 seconds.

Listing 9.11

```
try
   % general opening operations
   whichscreen = max(Screen('Screens'));
   [w, rect] = Screen('OpenWindow', whichscreen);
   % get the flip interval
   slack = Screen('GetFlipInterval', w)/2;

   % draw the fixation dot in the backgound
   Screen('FillOval', w, 0, CenterRect([0, 0, 10, 10], rect));
   % present the fixation
   fixation_onset = Screen('Flip', w); % get the onsettime of the fixation

   % draw the first stimulus in the backgound
   Screen('FillRect', w, [255, 0, 0], CenterRect([0, 0, 50, 50], rect));
   % present the first stimulus 0.5 seconds after the onset of the fixation
   firststimulus_onset = Screen('Flip', w, fixation_onset + 0.5 - slack);
   % keep the stimulus on screen for 0.75 seconds
   firststimulus_offset=Screen('Flip',w,firststimulus_onset + 0.75 -slack);

   % draw the second stimulus in the backgound
   Screen('FillRect', w, [0, 255, 0], CenterRect([0, 0, 50, 50], rect));
   % draw the second stimulus 1 second after the offset
   % of the first stimulus
   secondstimulus_onset=Screen('Flip',w,firststimulus_offset+1.0-slack);
   % keep the second stimulus on for 0.6 second
   secondstimulus_offset=Screen('Flip',w,secondstimulus_onset+0.6-slack);
   Screen('CloseAll'); % close all
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Analysis

In the example, the timing is achieved by getting the onset/offset times of each stimulus. For example:

On line 15 we flip the first stimulus at a time that is the sum of the onset of the fixation point and 0.5 s.

On line 17 we flip again after a period of 0.75 s. However, because there is nothing in the background, the screen is cleared after such a period. In this way, the presence of the first stimulus on the screen is controlled by this second flip.

On line 24 the second stimulus is switched 1 s after the offset time of the first stimulus.

On line 26 the second stimulus is cleared after 0.6 s due to the second flip.

Note the use of the subfunction 'GetFlipInterval' on line 6. This subfunction returns an estimate of the monitor flip interval for the specified onscreen window. This allows for maximum control of the display time. We use such slack in the computation of the "`when`" time in the flip subfunction.

Finally, there is another operation you can do to obtain maximal timing accuracy. When we use the computer there are several software processes running at the same time. The CPU does calculations for all of them. These activities reduce the resources that are available to MATLAB and PTB. PTB, however, has a set of functions for redirecting all available CPU resources to MATLAB to improve timing accuracy. These functions will be presented in the next chapter.

Summary

- The Psychophysics Toolbox (PTB) is a package specifically developed for psychophysics research.
- The core of the PTB is the `Screen` function.
- PTB uses a double buffering system (back and front buffers = 'background and foreground memory') that provides great timing control for visual stimuli.
- PTB can be used to draw objects (geometric figures, figures imported from graphics files or text) onscreen. The drawing of objects onscreen is performed in three steps: opening, drawing, and closing.
- The spatial arrangements of objects can be manipulated with the PsychRects functions, and the drawing can be done in batch-processing mode.
- The Screen subfunction flip can be used to control the timing of your stimuli and to synchronize the drawing with the screen's vertical retrace.
- The Screen function can be used to create movies.

Exercises

Exercise 1

Draw the Kanizsa triangle by designing three Pac-Men shaping the illusory contour.

Solution 1

```
try
    [w, rect] =Screen('Openwindow',0);
    cx = rect(3)/2;
    cy=  rect(4)/2;
    size = 90;
    displacement = 200;
    coord1 = [cx-size,cy-size-displacement,cx+size,cy+size-displacement];
    coord2 = [cx-size-displacement, cy-size+displacement/2,...
cx+size-displacement, cy+size+displacement/2];
    coord3 = [cx-size+displacement, cy-size+displacement/2,...
cx+size+displacement, cy+size+displacement/2];
    MystartAngle11 = 210;
    MystartAngle2 = 90;
    MystartAngle3 = 330;
    MyarcAngle = 300;
    Screen('FillArc', w, 0, coord1, MystartAngle, MyarcAngle);
    Screen('FillArc', w, 0, coord2, MystartAngle2, MyarcAngle);
    Screen('FillArc', w, 0, coord3, MystartAngle3, MyarcAngle);
    Screen ('Flip',w)
    mychar = KbWait;
    Screen('Close',w)
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

Exercise 2

Draw a Kanizsa triangle equal to the previous one but “dishonestly” (by shaping a white triangle on a white background whose vertex covers three black discs). Besides reviewing how to draw polygons, the purpose of this exercise is to realize that depending on where in your code listing you put a function, it can give rise to different results. Indeed, to solve this exercise, the triangle has to be drawn after, instead of before, the discs.

Solution 2

```
try
    [w, rect] =Screen('Openwindow',0);
    cx = rect(3)/2;
    cy=  rect(4)/2;
    size = 90;
    displacement = 200;
    coord1 = [cx-size, cy-size-displacement, cx+size, ...
              cy+size-displacement];
    coord2 = [cx-size-displacement, cy-size+displacement/2, ...
              cx+size-displacement, cy+size+displacement/2];
    coord3 = [cx-size+displacement, cy-size+displacement/2, ...
              cx+size+displacement, cy+size+displacement/2];
    coordtriangle =  [cx-displacement cy+displacement/2; ...
              cx cy-displacement; cx+displacement cy+displacement/2];
    Screen('FillOval', w, 0, coord1);
    Screen('FillOval', w, 0, coord2);
    Screen('FillOval', w, 0, coord3);
    Screen('FillPoly',w, 255,coordtriangle);
    Screen ('Flip',w)
    mychar = KbWait;
    Screen('Close',w)
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

A Brick for an Experiment

The stimulus for our experiment is simple: two discs that move with identical motions (one rightward, one leftward) and that start from one position and each end at the other disc's starting point. A stimulus similar to this has been shown previously in this chapter. Here we just reduce the size of the frame within which the discs are moving so that is a square of 300 by 300 pixels. The following script (slightly optimized in comparison to that of the chapter) shows the motion display used by Sekuler et al. (1997). In the example, two discs move: disc1 moves from left to right, disc2 from right to left.

```
try
   % opening operations
   whichscreen = max(Screen('Screens'));
   [w, rect] =Screen('Openwindow', whichscreen, 0);
   % discs setting
   frame = CenterRect([0, 0, 300, 300], rect);
   disc1 = AlignRect([0, 0, 20, 20], frame, 'center', 'left');
   disc2 = AlignRect([0, 0, 20, 20], frame, 'center', 'right');
   Screen('Flip', w);
   for i = 0:2:280
      Screen('FillRect', w, 255, frame);
      Screen('FillOval', w, 0, [disc1(1)+i,disc1(2),disc1(3)+i,disc1(4)
      Screen('FillOval', w, 0, [disc2(1)-i,disc2(2),disc2(3)-i,disc2(4)
      Screen('Flip', w);
   end
   Screen('FillRect', w, 255, frame);
   Screen('Flip', w);
   KbWait;
   Screen('CloseAll'); % close all
catch
    Screen('Close',w)
    rethrow(lasterror)
end
```

A few comments about this script. First of all, please note how we set the starting coordinates of all objects in a nested way. The coordinates of the frame within which the movement takes place are calculated according to the screen coordinates. The starting coordinates of the discs are calculated according to the coordinates of the frame. Note also that we flipped once before the for loop showing the motion. This simple operation enables us to synchronize the subsequent for loop (thus the successive flips) with the refresh rate. In a certain sense, we could say that we are "getting the pace" of the refresh rate. Note also how the i index is changed: in 2-unit steps. This increases the velocity of the motion (2 pixels per frame) in comparison to that originally shown in the chapter. The x coordinates of disc1 are increased by i (so that the discs moves toward the right), whereas the x coordinates of discs2 are decreased by i (so that the disc moves toward the left). At the end of the for loop, we again flip the frame only (but not the disc) so that the frame does not disappear after the motion.

We have now to build the second motion display, where the discs at the overlap point stop the motion for a few frames. Here we will stop the motion for two frames. In order to do this, we need the i index to remain for more than one cycle at the value 140. When i is equal to 140, the discs overlap completely. Everything can be done simply by modifying the beginning of the for loop as follows:

```
for i=[0:2:140, 140, 140:2:280]
```

Now the i index, the variable that lets us move the discs, increases from 0 to 140 (when the discs overlap), then is equal to 140, then increases from 140 to 280.

We can now write everything into the scripts and the function we wrote for the previous bricks. The opening and closing operations will be written into the SekulerExp function. The generation/presentation of the motion display will be written in a separate script that will be called MakeVideoStimulus.m. In this way we will not overload the text content of the SekulerExp function.

This is the script:

```
% discs setting
frame = CenterRect([0, 0, 300, 300], rect);
disc1 = AlignRect([0, 0, 20, 20], frame, 'center', 'left');
disc2 = AlignRect([0, 0, 20, 20], frame, 'center', 'right');

if VideoStimulusToPlay == 1
   Screen('Flip', w);
   for i = 0:2:280
      Screen('FillRect', w, 255, frame);
      Screen('FillOval',w,0,[disc1(1)+i,disc1(2),disc1(3)+i,disc1(4)]);
      Screen('FillOval',w,0,[disc2(1)-i,disc2(2),disc2(3)-i,disc2(4)]);
      Screen('Flip', w);
   end
elseif VideoStimulusToPlay == 2
   Screen('Flip', w);
   for i = [0:2:140, 140, 140:2:280]
      Screen('FillRect', w, 255, frame);
      Screen('FillOval', w,0,[disc1(1)+i,disc1(2),disc1(3)+i,disc1(4)]);
      Screen('FillOval', w,0,[disc2(1)-i,disc2(2),disc2(3)-i,disc2(4)]);
      Screen('Flip', w);
   end
end
Screen('FillRect', w, 255, frame);
Screen('Flip', w);
```

The following is the modified SekulerExp function. Note that we have written the KbWait command after the presentation of the stimulus.

```
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
EventTable = GenerateEventTable(conditions, repetitions, isfixed);
TotalNumberOfTrials = length(EventTable(:, 1));
% opening operations for the screen function
whichscreen = max(Screen('Screens'));
[w, rect] =Screen('Openwindow', whichscreen, 0);
refreshrate = Screen('FrameRate', w);
for trial = 1:TotalNumberOfTrials
    % STIMULI (SELECTION)
    VideoStimulusToPlay = EventTable(trial, 2);
    SoundStimulusToPlay = EventTable(trial, 3);
    % STIMULI (CREATION)
    % STIMULI (PRESENTATION)
    SoundToPlay = GenerateSound(SoundStimulusToPlay, w);
    MakeVideoStimulus
    % COLLECT SUBJECT'S ANSWER
    KbWait;
end
% closing operation
Screen('CloseAll'); % close all
% STORE RESULTS
```

In the next chapter we will play the sound and substitute the command KbWait with a more appropriate command that will enable us to get the participant's response. Moreover, we will see how to get maximal priority before running a movie, and finally, we will see how to get rid of the mouse pointer, which is unnecessary (and perhaps annoying) in the current experiment.

References

Brainard DH (1997) The psychophysics toolbox. Spat Vis 10:433–436
Kleiner M, Brainard DH, Pelli DG (2007) What's new in psychtoolbox-3? Perception (ECVP Abstract Supplement) 14
Pelli DG (1997) The VideoToolbox software for visual psychophysics: transforming numbers into movies. Spat Vis 10:437–442
Sekuler R, Sekuler AB, Lau R (1997) Sound alters visual motion perception. Nature 385:308

Suggested Readings

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

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