Part 2 – Introducing Psychtoolbox

What is Psychtoolbox?

Psychtoolbox is a set of Matlab functions (m-files) and Matlab executables (.mex files) written in C, which allow you to write simple code in order to generate scripts that will run psychological experiments.
Psychtoolbox has several advantages over other software used for this purpose: It is very flexible - you can create almost any stimulus you can imagine (and describe mathematically!). It has amassed a large and quite active community of users and developers. It is open source and is given free of charge. The support of the community is also free of charge and is, in most cases, quite superb. It is currently the only package of its kind that is documented in peer-reviewed publications:

Brainard, D. H. (1997) The Psychophysics Toolbox, Spatial Vision , 10:443-446.
Pelli, D. G. (1997) The VideoToolbox software for visual psychophysics: Transforming numbers into movies, Spatial Vision 10:437-442.

Don't forget to cite these good people when you publish results collected using PTB.
Nevertheless, PTB does have some distinct disadvantages: all the code is written by people just like you, in the middle of trying to do real science. This means that commands may not work as stated, help files may be out of date; commands may not even exist. Also, PTB runs on a commercial package (Matlab) and thus also suffers some of the problems of using commercial packages (closed code, price, etc…).
We will discuss alternative options at the end of the tutorial.

In order to start futzing with PTB, you need to download Psychtoolbox from here:

http://psychtoolbox.org/

You'll have to download Psychtoolbox from the web site. Make sure you download the right version (for MacOSX or PC). Once you have installed Matlab, then you can carry on with the chapter.

Basic graphics

Like I said before, Psychtoolbox enables you to display any stimulus you can describe mathematically. In this section we will review how to describe basic graphics and go over some of the tools that Matlab has in order to create graphics.

Colormaps

Begin with the following lines:

```
>>img = 1:5;
```

This is simply a vector with five elements.
Then put up a figure window in matlab:

```
>>figure(1)
>> paintpots1 =[0 0 0
    0.25 0.25 0.25
    0.5 0.5 0.5
    0.75 0.75 0.75
    1 1 1]
>>colormap(paintpots1);
```

I'll explain this colormap command in just a minute.

```
>>image(img)
```

The command `image(img)` draws a picture of the matrix `img`. In this case, `img` is a list of 5 numbers that is displayed as a row of vertical bars that gradually change in grayness as the value along the x-axis changes from 1 to 5.

In image the rows go along the x-axis from left to right and the columns go along the y-axis from the top to the bottom. So try this:

```
>>image(img')
```

Now the bars will be horizontal and will increase in brightness as you go from the top to the bottom.

```
>>axis off
>>axis square
```

This will get rid of the axes

```
>>paintpots2=[0 0 1; 1 0 0; 0 1 0; 0.5 0 1; 1 0 1];
>>colormap(paintpots2)
```

Whoah! Why does the image suddenly change color?

`paintpots` represent two different sets of paints.

In `paintpots1` the values 1-5 was represented by different levels of gray. The command to tell Matlab to paint the image img using paintpots1 was `colormap(paintpots1)` . Basically a ***`colormap`*** is a *mapping* between a set of indices in an image (your paint-by-numbers picture) and a set of *colors* (your picture). You may also hear the word ***palette*** to describe a colormap – the two words mean the same thing.

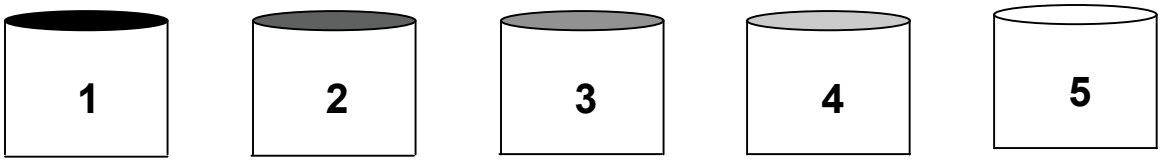
If you look at paintpots1 you will see that it is a 5x3 matrix.

```
>>size(paintpots1)
```

The 5 rows refer to the fact that paintpots has five paint pots (indices into the colormap).
The 3 rows refer to the intensity of the red, green a blue gun. These gun intensities go between 0 (black) and 1 (white).

```
>>paintpots1
```

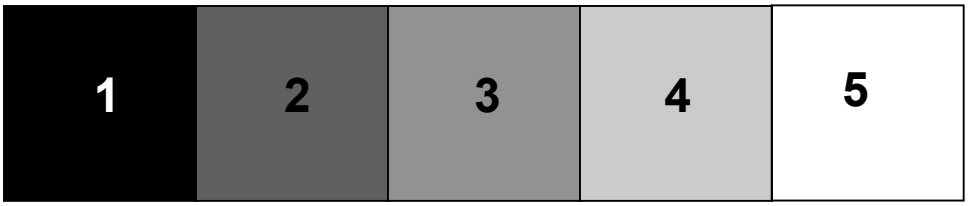
So the first paint pot contains black, the fifth paint pot contains white, and the intermediate ones contain gradually lighter colors of gray.



This is why

```
>image(img)
>colormap(paintpots1)
```

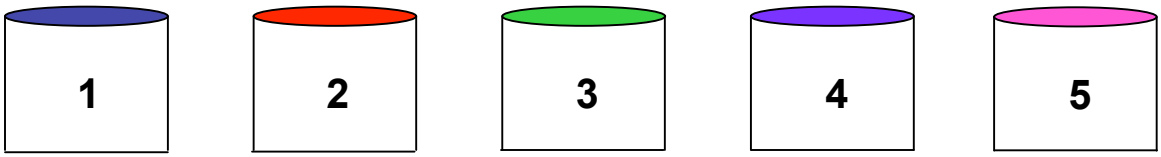
produces an image that gradually goes from black to white.



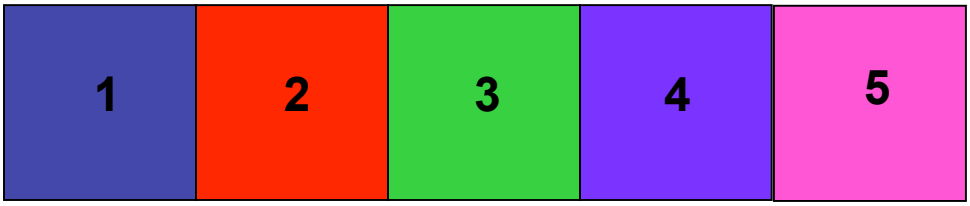
Now lets look at paintpots2. Once again, paintpots2 represents 5 different colors.

```
>paintpots2

paintpots2 =
         0      0      1   % only the blue gun, at maximum, this makes blue
         1      0      0   % only the red gun, at maximum, this makes red
         0      1      0   % only the green gun, at maximum, this makes green
       0.5      0      1   % some red and lots of blue, makes purple
         1      0      1   % red and blue, makes pink
```



So when you paint the img using this set of paintpots (a new colormap) the image changes.



One cute thing about Matlab is that it has a lot of pre-made colormaps. Check out:

```
>>paintpots3=hot(5);
>>colormap(paintpots3);
>>paintpots4=hsv(5);
>>colormap(paintpots4);
```

Also check out `winter, spring, cool, jet.`
You can find a full list of Matlab colormaps using

```
>>help graph3d
```

Spend some time playing with colormaps. For example, see if you can change a single row of a colormap. For example if you want to replace purple with yellow:

```
>>paintpots2b(4, :)=[1 1 0]
```

You should be able to guess what will happen if you use paintpots2b as your colormap.
Now see if you can remove all the red from paintpots1.
Now we are going to look at a little m-file, which will allow you to gradually change an image by changing the colormap:

```
1       % UsingColormaps.m
2       %
3       % a little program that manipulates colormaps
4       %
5       % written by IF and GMB 3/2005
6
7       clear all
8       close all
9       img=1:10;
10      figure(1)
11      paintpots = ones(10,3);
12      colormap(paintpots)
13      image(img);
14      axis off;
15      for i=1:10
16          paintpots (i,:)= (i/10);
17          colormap(paintpots);
18          pause
19      end
```

Lines 15-19. As you go from i = 1 to 10, you are gradually replacing the 1's in paintpots with grays that go from 0.1 (very dark) to 1 (white). Then you replace the old colormap with the new colormap, and gradually the white in the image is replaced by grays. .

Line 18. The pause makes the program wait for a key press so you can observe each step.
Now we are going to replace lines 3-13 in UsingColormaps.m

```
1       % UsingColormaps2.m
2       %
3       % Another little program that manipulates colormaps
4       %
5       % written by IF and GMB 3/2005
6
7       clear all
8       close all
9
10      colormap(gray(256))
11      img = reshape(1:256,16,16);
12      image(img);
13      axis square
14      axis off
15      pause
16      for i=1:200
17          paintpots = rand(256,3);
```

```
18          colormap(paintpots);
19          drawnow
20      end
```

I hope that was at least just a little bit exciting!

Line 11: The command ***reshape*** takes 3 arguments and allows you to reshape a vector or matrix into a different shape. The first argument is the vector or matrix that you want to reshape. In this case we give reshape a vector that goes [1 2 3 …256]. The second argument is the number of rows you want the new matrix to have, and the second argument is the number of columns you want the new matrix to have. So in this case reshape turns a 1x256 vector into a matrix with 16 rows and 16 columns. Of course this only works because 16x16=256. Otherwise you would get the following error

```
>>img = reshape(1:254,16,16);
??? Error using ==> reshape
To RESHAPE the number of elements must not change.
```

Take a look at img

```
>>img
```

Here's another example of using reshape:

```
>>tmp=[3 4 4 5 1 2; 6 7 1 2 2 3]
>>tmp2=reshape(tmp, 3, 4)
>>tmp3=reshape(tmp, 12, 1)
```

Lines 12-14: simply draws the matrix img, removes the axis and makes the image square. Waits for a keypress.
Lines 16-20: i will step from 1, 2, 3 …200. Each time, lines 17-19 will be executed. You remember how a `colormap` is like a collection of paint pots – with each paint pot having a number label. Here we have 256 paint pots, and the colors inside the paint pot are being assigned randomly. `rand(256,3)` creates a 256 x 3 matrix of random numbers. The random numbers vary between 0-1, so that allows us to define 256 random colors to go into the paint pots.
(A brief digression: if you type just the command `rand` then Matlab will just give you a single randomly chosen number from a uniform distribution. Otherwise you have to describe the size of the matrix of random numbers that you want, as is done here. Try `help rand` for more details)
Line 19: Tells Matlab to update the figure. Normally updating figures have a pretty low priority for Matlab, `drawnow` tells Matlab to put the other calculations on hold until it's updated the figure.

Making a Gabor

Here we are going to use some of the things we have learned to make a Gabor filter. These are regularly used for image processing. They are also popular as stimuli among vision scientists.

```
>>sd=.3;
>>x=linspace(-1,1,100);
>>y=(1/sqrt(2*pi*sd)).*exp(-.5*((x/sd).^2));
>>y=y./max(y);
```

So `y` is a 1-d Gaussian with a standard deviation of `sd=0.3.` It has a minimum value of 0 and a maximum value of 1. We can `plot` this and see what it looks like.

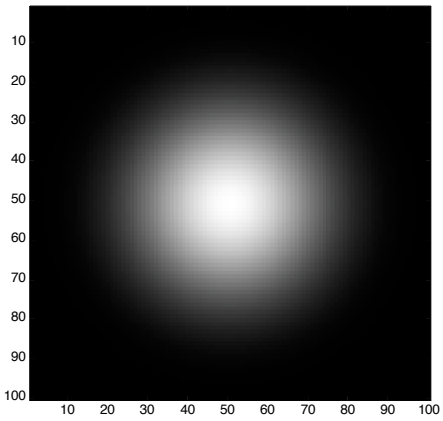
```
>>plot(x, y);
```

Now we use the outer product to create a two-dimensional Gaussian filter, with a maximum value of 1

```
>>filt=(y'*y);
>>max(filt(:))
```

And then we can look what that two dimensional Gaussian looks like. We are going to allow the colormap to take 256 possible values of gray. That means we want filt to vary between 1 and 256.

```
>>colormap(gray(256));
>>scaled_filt=scaleif(filt, 1,256);
>>image(scaled_filt)
```

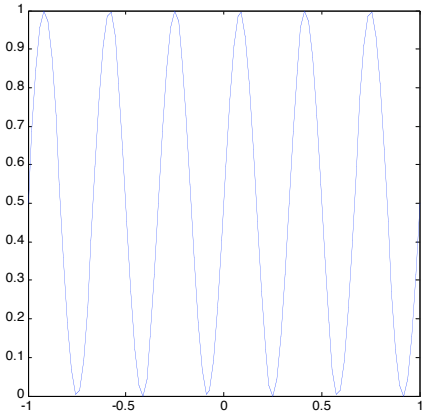


We can then use this Gaussian window to filter any image we want. Let's say we want to filter a sinusoidal grating. We actually use much the same set of tricks.

```
>>sf=6; % spatial freq in cycles per image
>>y2=sin(x*pi*sf);
>>y2=scaleif(y2, 0, 1);
```

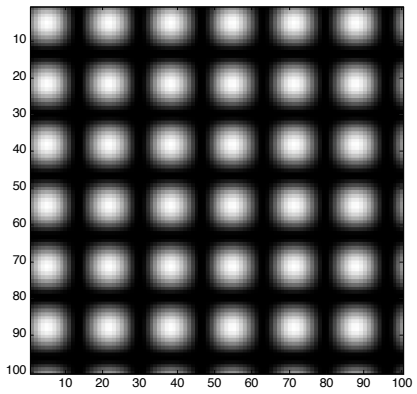
scaleif is a custom-made function, which you can download from the tutorial website.Wonder what it does? Type `'help scaleif'`

```
>>plot(x, y2);
```



This now gives us a 1-dimensional sinusoidal grating with a frequency of 6 cycles per image. We've scaled it so it has a minimum value of 0 and a maximum value of 1. If we calculate the outer product of this sinusoid with itself we get a weird checkerboard.

```
>>img=(y2'*y2);
>>colormap(gray(256))
>>scaled_img=scaleif(img, 1, 256)
>>image(scaled_img)
```

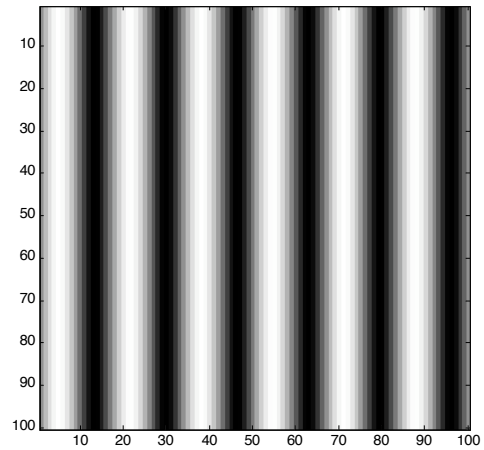


What we want to do is calculate the outer product of the one-dimensional sinusoid with a vector of ones.

```
>>y3=ones(size(y2));
>>img=(y3'*y2);
>>colormap(gray(256))
>>scaled_img=scaleif(img, 1, 256)
>>image(scaled_img)
```

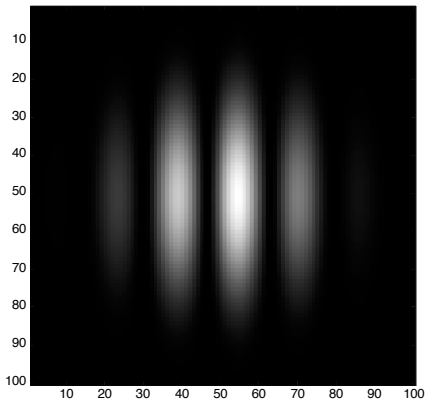
Once again, img has a minimum of 0 and a maximum of 1

```
>>max(img(:))
```



Now to create a Gabor (a sinusoid windowed by a Gaussian) becomes a piece of cake. We simply multiply the two-dimensional Gaussian window by the two-dimensional Gabor.

```
>>gabor=img.*filt;
>>scaled_gabor=scaleif(gabor, 1, 256);
>>colormap(gray(256))
>>image(scaled_gabor);
```



Note that throughout this process we have worked with two versions of the Gaussian and the Gabor. The original versions (`filt, img`) were scaled between 0 and 1. We did this to keep the nice property that when the Gaussian filter was at its maximum, the brightness of the grating didn't change, and when the Gaussian filter was at 0 the Gabor was also at 0. But when we used `image` to look at these matrices we converted them to range between 1-256 so as to match the colormap.

This gives you some basic stuff to start with.
Again – the limit of making visual stimuli is your imagination and your ability to describe what you imagine in numbers.

Getting started with PsychToolbox Screen.m

In order to know if PTB is properly installed try running the following:

```
>>ScreenTest
```

If you get the screen going blank and then something like the following then `screenTest` worked

```
***** ScreenTest: Testing Screen 0 *****


PTB-INFO: This is the OpenGL-Psychtoolbox version 3.0.8. Type
'PsychtoolboxVersion' for more detailed version information.
PTB-INFO: Psychtoolbox is licensed to you under terms of the GNU
General Public License (GPL). See file 'License.txt' in the
PTB-INFO: Psychtoolbox root folder for a copy of the GPL license.


PTB-INFO: OpenGL-Renderer is NVIDIA Corporation :: GeForce Go
7400/PCI/SSE2 :: 2.0.1
PTB-Info: VBL startline = 768 , VBL Endline = -1
PTB-Info: Measured monitor refresh interval from VBLsync =
16.712593 ms [59.835118 Hz]. (50 valid samples taken,
stddev=0.044112 ms.)
PTB-Info: Reported monitor refresh interval from operating system
= 16.666667 ms [60.000000 Hz].
```

```
PTB-Info: Small deviations between reported values are normal and
no reason to worry.
PTB-INFO: Using NVidia's GL_TEXTURE_RECTANGLE_NV extension for
efficient high-performance texture mapping...

***** ScreenTest: Done With Screen 0 *****
```

If not, then Psychtoolbox isn't working right on your computer

Writing code using Screen.m

For now, let's go back to the basics. We are simply going to open an experimental window, making it black, making it white, and then closing it again.

The script below uses a command called `Screen`, which is one of the core functions of Psychtoolbox. Screen is actually not an m file, like the ones you have probably been writing in Matlab. It is a mex file. This means that it is written in C (or C++) or some other programming language, and is then compiled to run in Matlab. The reason this was done is because Screen does some pretty funky stuff that would be impossible in Matlab.

```
1   % DarkScreen.m
2   %
3   % opens a window using Psychtoolbox,
4   % makes the window black, then white, and then closes
5   % the window again
6   %
7   % written for Psychtoolbox 3 on the PC by IF 3/2007
8
9   screenNum=0;
10  res=[1280 1024];
11  clrdepth=32;
12  [wPtr,rect]=Screen('OpenWindow',screenNum,0,...
13  [0 0 res(1) res(2)], clrdepth);
14  black=BlackIndex(wPtr);
15  white=WhiteIndex(wPtr);
16  Screen('FillRect',wPtr,black);
17  Screen(wPtr, 'Flip');
18
19  HideCursor;
20  tic
21  while toc<3
22      ;
23  end
24
25  Screen('FillRect',wPtr,white);
26  Screen(wPtr, 'Flip');
27  HideCursor;
28  tic
29  while toc<3
30      ;
31  end
32
33
34  Screen('CloseAll');
35  ShowCursor
```

Line 9. If you are running more than one monitor on your computer then Screen tells you which one to use. 0 means the monitor with the menu bar (or in Mac OSX, the dock) 1 means the other monitor. For now, if you have any difficulty I would make sure you are only using one monitor. If you are cloning your monitor then Screen 1 becomes the display screen.
Line 10-11. Check the resolution of your screen and the using. As far as the resolution of the monitor in pixels is concerned the first number is the width of the screen, the second number is the height of the screen. The resolution of your screen in terms of color depth will be 8, 16 or 32 bits, depending on the age of your monitor or computer. Set it to the highest setting the computer will allow.
Line 12. This calls a function called Screen, which takes 5 arguments.
Argument 1. a command telling Screen what to do – in this case you want Screen to open a window. *Argument 2*. which monitor you want the window opened inside. In this case you want the Screen opened in monitor 0 – the one with the menu bar.
Argument 3. The color you want to fill the window with. This currently doesn't seem to work in Psychtoolbox.
Argument 4. This tells Screen how big you want the window to be – in this case you want it to be a rectangle the size of the entire screen. The order in which you describe this rectangle can be remembered as **L**e**T**te**RB**ox (Left, Top, Right, Bottom). We want the rectangle to start 0 pixels from the Left, and 0 pixels from the Top, and go to 1280 pixels towards the Right and 1024 pixels towards the Bottom.
(Why does it start with 0 instead of 1? Because Screen is a mex file and was written in C which is a 0-based language.)
Currently Psychtoolbox always uses the whole screen, no matter what you enter into rect. *Argument 5* is the color depth of the monitor.
Arguments 3-5 don't actually need to be specified – Matlab will default to certain values: a random background color, the whole screen and the color depth of the monitor as described in the control panel
Lines 14 - 15 Find the colormap values that will give you black and white.
Line 16 Draw a black rectangle the size of the screen. Psychtoolbox automatically assumes that you have an offscreen window and an onscreen window. Every time you use a drawing command like DrawRect it will automatically draw on the offscreen window. So the rectangle won't yet be visible. The advantage of this approach (as you will see later) is that you can spend some time drawing several things offscreen without them showing up one by one on the screen. Once you have finished drawing you can move the offscreen window to the front.
Line 17 You need to `Flip` the screen so the offscreen window you drew the black rectangle on comes to the onscreen window – the one that is actually on the monitor.
Monitors have a ***refresh*** rate (of something between 60Hz-120Hz for a standard monitor). Every refresh begins at the top of the screen, and moves quickly down the screen in a matter of a few milliseconds. The flip command simply tells Matlab to refresh the screen with the image you have drawn offscreen.

Lines 19-23 Hide the cursor and wait 3 seconds
Lines 25-31 Draw a white rectangle on the offscreen window, flip it to the front and wait 3 seconds
Lines 34-35. Close the screens and re-appear the cursor

One weird thing about Screen is that you don't get help about it's commands in the normal way.

If you type:

```
>>help Screen
```

You get a lot of info, but it doesn't really tell you how to use `Screen`.
If you try:

```
>>Screen
```

You will get a list of all the subcommands that are contained within Screen. To get more information about a particular command you do:

```
>Screen OpenWindow?
>Screen DrawText?
```

****

NOTE FOR IF YOUR SCREEN FREEZES
If you display stimuli on the main screen, as we often do, then the Screen window will hide the main menu bar and obscure Matlab's command window. That can be a problem if your program stops (perhaps due to an error) before closing the window. The keyboard will seem to be dead because its output is directed to the front-most window, which belongs to Screen not Matlab, so Matlab won't be aware of your typing.

Remain calm.

Typing Ctrl-C will stop your program if hasn't stopped already. Typing:

command-zero (on the Mac)
Alt-Tab (on Windows)

Will bring Matlab's command window forward. That will restore keyboard input.
Typing:

```
>>sca
```

will call Screen('CloseAll'). This will close all the open screens and will cause the cursor to reappear (if you hid it)
The screen might still be hard to make out, if you've been playing with the lookup table.
Typing:

```
>>clear Screen
```

Should flush all the changes made by Screen from memory and restore your screen to normal. If that doesn't work, do the usual dance – quit Matlab and restart it. If all else fails, throw away your pc and get a mac.
****

Here's another more elaborate example of how you can use `Screen`

```
1    % FunkyScreen.m
2    %
3    % opens a window using psychtoolbox,
4    % makes the window do some funky things
5    %
6    % written for Psychtoolbox 3 on the PC by IF 3/2007
7
8    screenNum=0;
9    flipSpd=13; %  a flip every 13 frames
10
11   [wPtr,rect]=Screen('OpenWindow',screenNum);
12
13
14   monitorFlipInterval=Screen('GetFlipInterval', wPtr);
15   % 1/monitorFlipInterval is the frame rate of the monitor
16
```

```
black=BlackIndex(wPtr);
white=WhiteIndex(wPtr);

% blank the Screen and wait a second
Screen('FillRect',wPtr,black);
Screen(wPtr, 'Flip');
HideCursor;
tic
while toc<1
    ;
end

% make a rectangle in the middle of the screen; flip colors and size
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip'); % collect the time for the first flip with vbl
for i=1:10
    Screen('FillRect',wPtr,[0 0 255], [100 150 200 250]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
% flip 13 frames after vbl
    Screen('FillRect',wPtr,[255 0 0], [100 150 400 450]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
end

% blank the screen and wait a second
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
tic
while toc<1
    ;
end

% make circles flip colors & size
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip');
for i=1:10
    Screen('FillOval',wPtr,[0 180 255], [ 500 500 600 600]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
    Screen('FillOval',wPtr,[0 255 0], [ 400 400 900 700]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
end

% blank the Screen and wait a second
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
tic
while toc<1
    ;
end

% make lines that flip colors size  & position
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip');
for i=1:10
    Screen('DrawLine',wPtr,[0 255 255], 500, 200, 700 ,600, 5);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
    Screen('DrawLine',wPtr,[255 255 0], 100, 600, 600 ,100, 5);
```

```
74      vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
75  end
76
77  % blank the Screen and wait a second
78  Screen('FillRect',wPtr,black);
79  vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
80  tic
81  while toc<1
82      ;
83  end
84
85  % combine the stimuli
86  Screen('FillRect',wPtr,black);
87  vbl=Screen(wPtr, 'Flip');
88  for i=1:10
89      Screen('FillRect',wPtr,[0 0 255], [100 150 200 250]);
90      Screen('DrawLine',wPtr,[0 255 255], 500, 200, 700 ,600, 5);
91      Screen('FillOval',wPtr,[0 180 255], [ 500 500 600 600]);
92      Screen('TextSize', wPtr , 150);
93      Screen('DrawText', wPtr, 'FUNKY!!', 200, 20, [255 50 255]);
94      vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
95
96
97      Screen('FillRect',wPtr,[255 0 0], [100 150 400 450]);
98      Screen('FillOval',wPtr,[0 255 0], [ 400 400 900 700]);
99      Screen('DrawLine',wPtr,[255 255 0], 100, 600, 600 ,100, 5);
100     vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
101 end
102
103 % blank the screen and wait a second
104 Screen('FillRect',wPtr,black);
105 vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
106 tic
107 while toc<1
108     ;
109 end
110
111 Screen('CloseAll');
112 ShowCursor
113
114
```

Line 9. Here you define `flipSpd`, you are going to make the display flip every 13 frames.
Line 11. This time when you open the monitor you let the monitor choose the resolution and the color depth. The monitor actually returns the resolution in rect.
Line 13 You can find out how fast your monitor flips using `'GetFlipInterval'`
Line 31. This time when you flipped the window you made `Screen` return the time (according to the computer clock) that it did the flip. That time is saved as `vbl`. VBL stands for "Vertical Blanking", which is a synonym of the vertical retrace of the screen by the electron beam that draws the screen display.
Line 33. Here we are using `FillRect` again, but this time we are defining the rect as only being a subset of the entire screen. Remember that the `rect` is defined as LeTteRBox. The other difference is that this time, instead of sending it a single number that is an index into the colormap, we are sending it 3 values - for the red, green and blue guns. There is a weirdness here that the monitor thinks that it is a 32 bit monitor, but these values are on an 8 bit scale. This is because allowing the red gun to take any number between 0-255 takes up 8 bits, allowing the green gun to take any number between 0-255 takes up 8 bits, allowing the blue gun to take any number between 0-255 takes up 8 bits. 3x8 is 24. The other 8 bits are padding – don't ask me what they are used for.

Line 34: We `flip` the screen, but this time we are telling it to flip at time vbl + (13 * monitorFlipInterval). This means it will flip 13 frames after the flip on line 31. Once again the time that the monitor was flipped is saved as vbl.
Lines 36-37. We do the same thing again, but this time `vbl` refers to the flip that happened on line 34. So once again there is a 13 frame wait before the flip.
Line 53. `FillOval` works just like `FillRect` except that it draws ovals instead of squares.
Line 71. `DrawLine` doesn't take a `rect` as an argument. Instead it takes in the starting horizontal position, starting vertical position, ending horizontal position, ending horizontal position. I guess the mnemonic for that would be **H**um**V**ee
Lines 89-92 Here we are drawing more than one thing on the offscreen window before we flip them to the front
Lines 93-94. Here we are drawing text on the screen. First we define the size of the text as having a font size of 150. Then we draw it on the screen. 200, 20 refer to the starting horizontal and vertical positions of where you want the text placed (**H**um**V**ee again). `[255 50 255]` refers to the color of the text.

PsychDemos

If you are wondering what kind of thing you can do with PTB and how some of your ideas can be coded up, look at the demos that ship with PTB. Try typing:

```
>>help PsychDemos
```

This will give you a list of available demos and a short description of what they do. If you are curious what a certain demo does you can inquire further. For example, type:

```
>>help MandelbrotDemo
```

This will tell you what this script does. If you are curious how this is implemented, type:

```
>>edit MandelbrotDemo
```

This will open the file MandelbrotDemo.m in an editor window. Don't edit this file! You might cause some damage. Instead, save the file under a new name. For example, 'myMandelbrotDemo.m'. Now you can twiddle things in the file and try to see what effect these changes have on the execution of the program. But before you start doing that, let's get acquainted with the single most important function in Psychtoolbox.

Key Presses

In most experiments that we do presenting the stimulus is only half of the action. Collecting responses from our subjects is key (har, har)! Here we will go through the various ways that you can collect keypresses and discuss their strengths and weaknesses. Different techniques are useful for different purposes. This table should help you keep these commands straight in your head.

	Psychtoolbox windows	Timing	What was pressed	Waits for keypress	NOTES
`Pause`	NO	NO	NO	YES	Just waits for a specified period of time or for any key to be pressed on the keyboard
`Input`	NO	NO	YES (Matlab	YES	Just waits for a key to be pressed on the

			expression)		keyboard and returns the character or number of that key
`CharAvail`	YES	NO	NO	NO	Simply checks whether there is a key press in the event queue
`GetChar`	YES	Dubious accuracy	YES (character)	YES	Checks or waits for a key press in the event queue. Returns what the key was, and when it was pressed
`KbCheck`	YES	Good	YES (key)	NO	Tests whether a key has been pressed at that moment in time
`KbWait`	YES	Good	NO	YES	Waits for a key press to occur

Collecting keypresses using pause and input

These techniques are useful when you are not using Psychtoolbox. Remember that none of these techniques have good timing. It's easiest practicing these commands in an m-file, since many of these commands will accept Return as a key press (if you type them into the command line, the Return you use at the end of the line will also be used as the input into the key press command).

pause

This is the simplest command you can use to collect a key-press. It simply pauses the computer until a key (any key) is pressed on the computer and doesn't collect what that response is.
Bear in mind that the command window needs to be in front for the keypress to be available to Matlab, so pause doesn't work well if you are using Psychtoolbox. In that case use `GetChar` instead.
Pause can also be used to enforce a delay –

```
>>pause(3)
```

pauses for 3 seconds.

input

This command again pauses the computer until a key is pressed but it allows you to collect the response. By default the response is a number

```
>>resp_num=input('press a number key ...')
```

But you can also specify that `input` will accept a string, as shown below. In that case if the subject inputs a number key the computer will assume that the subject chose the character '3' rather than the double 3.

```
>>resp_char=input('press a number key ...', 's')
>>whos
```

Note that `resp_num` is a double and `resp_char` is a character.

If in a program you want to only accept a certain type of response, then you need to write a loop like this one:

```
1       % PracticeKeyPresses
2       % a program to practice different ways of collecting
3       % key press responses
4       %
5       % written if 4/2007
6
7       % using input
8       disp('Using input command')
9       resp='x';
10      while resp~='a' & resp~='b'
11          resp=input('press a or b ... ', 's');
12      end
13      resp
14
```

Bear in mind that the command window needs to be in front for the key-press to be available to Matlab, so `input` doesn't work well if you are using Psychtoolbox. In that case you will need to use `GetChar`, `CharAvail` or `KbCheck` instead.

Collecting keypresses using GetChar & CharAvail

When you type into the keyboard (or any input device) the key presses get saved into an event queue. This is why sometimes when you type really fast and the computer is also busy with other tasks there will be a pause before the letters suddenly appear in your document. `CharAvail` and `GetChar` commands read from the event queue. It's therefore important to empty events from the event queue before collecting responses using these two commands

```
>>FlushEvents
```

Before using any of the following commands.

```
>>CharAvail
```

looks to see if there is anything in the event queue.

```
[avail, numChars] = CharAvail;
```

`avail` will be 1 if characters are available, 0 otherwise. `numChars` may hold the current number of characters in the event queue, but in some system configurations it is just empty, so do not rely on `numChars` providing meaningful results, unless you've tested it on your specific setup. If `avail` is 1 then you need to call `GetChar` to find out what the actual key press is – `CharAvail` just tells you whether there was a key press.

```
15      % CharAvail
16      WaitSecs(1)
17      disp('Using CharAvail command')
18      disp('Wait 2 sec to see if you pressed a key during that
19      time');
20      FlushEvents
21      tic
22      while toc<2
23       ;
24      end
25      resp = CharAvail;
```

```
26      resp
```

CharAvail is useful when you want to see whether your subject pressed a key while you were doing something else (such as displaying images) in the meantime. It's especially useful if you are doing something like fmri where you don't want to wait indefinitely for a subject response.

GetChar

If there is something already in the event queue then `GetChar` retrieves it. If there isn't something in the Event queue then GetChar waits for a typed character and lets you save the response.

It's used as follows:

```
[resp, when] = GetChar([getExtendedData], [getRawCode]);
```

`char` is the character that was typed
`when` is a structure. It returns the time of the key press, the "adb" address of the input device, and the state of all the modifier keys (shift, control, command, option, alphaLock) and the mouse button. If you have multiple keyboards connected, address may allow you to distinguish which is responsible for the key press. `when.secs` is an estimate of the time when the key press happened. However the timing of `GetChar` is not necessarily reliable, the reported values can be off by multiple dozen or even hundreds of milliseconds. If you are interested in precise timing you should check the timing of `GetChar` on the particular system that you are using or use `KbWait` or `KbCheck` instead.

`getExtendedData` tells `GetChar` whether or not to collect when data, if set to 0 then `GetChar` will be a tiny bit faster. `getRawCode` determines whether you want to return the character information in ascii or char (the default) format. Normally you won't bother using either of these input arguments.

Add the following lines to your `PracticeKeyPresses` m-file. If you press a key GetChar behaves very like CharAvail except that it retrieves what the character is. If you don't then GetChar waits

```
27      %Using GetChar
28      WaitSecs(1)
29      disp('Using GetChar command just on its own')
30      FlushEvents
31      tic
32      while toc<1
33          ;
34      end
35      [resp_GC, when_GC]=GetChar;
36
37      disp('now you have pressed a key')
38      resp_GC
39
```

Note that even though the command line may appear, the program doesn't actually continue to print `'now you have pressed a key'` until you press a key – if there's nothing in the event queue then `GetChar` will wait for a response.

What if you only want to accept certain responses (e.g. the 'm' or 'f' keys)?

```
40      %GetChar - only accepting certain responses
41      WaitSecs(1)
```

```
42      disp('Using GetChar command - only accepting certain
43      responses')
44      FlushEvents
45      resp_GC2='x';
46      while resp_GC2~='m' & resp_GC2~='f'
47          disp('press m or f ... ');
48          resp_GC2=GetChar;
49      end
50      resp_GC2
```

What if you want to combine the useful property of CharAvail that it doesn't wait indefinitely for a response with the useful property of GetChar that you can find out what the response was?

```
51      % Combining GetChar and Char Avail
52      WaitSecs(1)
53      disp('Combining GetChar and Char Avail')
54      disp('Wait 2 sec to see if you pressed a key during that
55      time');
56      FlushEvents
57      tic
58      while toc<2
59          ;
60      end
61      resp_CA3= CharAvail;
62      if resp_CA3==1
63          resp_GC3=GetChar;
64          disp(['You pressed key', resp_GC3])
65      else
66          disp('You did not press a key');
67      end
```

One more thing to remember

KbCheck and KbWait are MEX files, which take time to load when they're first called. They'll then stay loaded until you flush them (e.g. by changing directory or calling CLEAR MEX). So your timing on the first trial will be more accurate if you just call them at the beginning of the program.

Collecting keypresses using KbCheck & KbWait

`KbCheck` and `KbWait` don't pull key presses out of the event queue. Instead they monitor the state of the keyboard at that very moment. The advantage of this is that they tend to have more accurate timing on some systems, and there is no lag – the minute the key press happens the `KbCheck` and `KbWait` command knows that the key press has happened. The disadvantage is that it's hard to do something else (e.g. display images) while constantly monitoring the state of the keyboard.

One thing that is a little weird about `KbCheck` and `KbWait` is that they actually refer to the key that was pressed on the keyboard rather than the character represented by that key. This is because they are lower level commands. What character a key represents actually varies across computer systems, which means you need to convert your key press information into character information.

While most keynames are shared between Windows and Macintosh, not all are. Some key names are used only on Windows, and other key names are used only on Macintosh. For a lists of key names common to both platforms and unique to each see the comments in the body of KbName.m.

KbName will try to use a mostly shared name mapping if you add the command KbName('UnifyKeyNames'); at the top of your experiment script. In fact, **KbName often won't work unless you add this line to the beginning of your scripts!**

Psychtoolbox has a great demo for KbCheck and KbWait which is called KbDemo. You can try it by just typing KbDemo at the command window:
>KbDemo
This is also a good way of making sure that KbCheck is working. If not, try replacing the version of KbCheck.m that you have with a revised version found here:

http://en.wikibooks.org/wiki/Psychtoolbox:KbCheck

The examples below are simplified versions of taken from KbDemo

KbCheck

Checks to see whether a key on the keyboard is pressed down at that very moment in time.

```
[keyIsDown,secs,keyCode] = KbCheck;
```

`keyIsDown` is 1 if *any* key, including modifiers such as <shift>,<control> or <caps lock> is down. 0 otherwise.

`secs` is the time of the key press as returned by `GetSecs`. This is an accurate way of getting timing but you should still read the help file for `KbCheck` and `GetSecs` carefully if your experiment depends on accurate timing.

`keyCode` On Macs this is a 128-element array, on PCs it is a 256 . Each number in the array represents a keyboard key. If a key is pressed that index in the array is set to 1, otherwise it will be set to 0. To convert a `keyCode` to a vector of key numbers that were pressed use `find(keyCode).` To find out what actual key that was use `KbName`.

KbWait

Just like KbCheck, but it waits until a key on the keyboard is pressed down and simply returns the time that the key was pressed.

```
secs = KbWait;
```

`secs` is the time of the key press as returned by `GetSecs`.

KbName

KbName maps between KbCheck-style keyscan codes and the character that key represents. Use KbName to make your scripts more readable and portable, since keycodes, are cryptic and vary between Mac and Windows computers.

```
kbNameResult = KbName(arg)
```

KbName actually will let you go either way from keycodes to characters, or vice versa. If you send in as input a string designating a character then KbName returns the keycode for that character.

```
>KbName('t')
```

If on the other hand you send in a number representing the keycode, then KbName will return the character of that keycode

```
>KbName(84)
```

The numbering of these practice examples matches their order in KbDemo, but I'm going through them in order of how complicated they are. That's why the naming has a funny order (we start with KbPractice3)

```
1       % exampleKb
2       % Wait for a key with KbWait.
3       % a simplified version of KbDemo by IF 4/2007
4
5       WaitSecs(0.5);
```

```
6       disp('Testing KbWait: hit any key.  Just once.');
7
8       startSecs = GetSecs;
9       timeSecs = KbWait;
10
11      [keyIsDown, t, keyCode ] = KbCheck;
12
13      str=[KbName(keyCode), ' typed at time ', …
14      num2str(timeSecs - startSecs), ' seconds'];
15      disp(str);
16
```

Line 8-10. KbWait returns the time in seconds (with high precision) since the computer started. Usually we don't want to know the absolute time in seconds, but the time since some other event (e.g. since you put an image on the screen). So here we are calculating the time between the key press and an arbitrary start time.

```
1       %exampleKb2.m
2       % Displays the key number when the user presses a key.
3       % a simplified version of KbDemo by IF 4/2007
4
5       WaitSecs(0.5);
6       disp('Testing KbCheck and KbName: press a key to see its
7       number');
8       disp('Press the escape key to exit.');
9       escapeKey = KbName('ESCAPE');
10
11      while KbCheck
12      ;
13      end % Wait until all keys are released.
14
15      while 1
16          % Check the state of the keyboard.
17          [ keyIsDown, seconds, keyCode ] = KbCheck;
18
19      % If the user is pressing a key,
20      % then display its code number and name.
21          if keyIsDown
22
23      % Note that we use find(keyCode) because keyCode is an array.
24                  str=['You pressed key ', find(keyCode), …
25      ' which is ', KbName(keyCode)];
26                  disp(str)
27
28                  if keyCode(escapeKey)
29                      break;
30                  end
31
32      % If the user holds down a key,
33      % KbCheck will report multiple events.
34      % To condense multiple 'keyDown' events into a single event,
35      % once a key has been pressed
36      % we wait until all keys have been released
37      % before going through the loop again
38                  while KbCheck; end
39          end
40
```

```
end
```