Keith Schneider, 27 July 2006

Matlab and Psychophysics Toolbox Seminar

Part 2. Psychophysics Toolbox and Static Images

The Psychophysics Toolbox (PTB) is a set of compiled (known as mex files) functions for Matlab. The newest version utilizes OpenGL graphics functions that take advantage of the graphics processing power of modern video cards. Consequently, it is significantly faster and more versatile than the previous versions, and programming tricks such as color table animation that were used in the past because of speed limitations are no longer necessary. This new version works with Mac OSX or Windows machines, and maybe soon Linux. If you have an old script written for a previous version of the PTB (Mac OS9), you can still probably run them under the new PTB by inserting the line `Screen('Preference','EmulateOldPTB',1)` at the top of your old script. New experiments should be written with the new PTB.

The Psychophysics Toolbox includes a number of demo programs and tests. For a list, go to the `Psychtoolbox/PsychDemos` folder and type `ls` or `dir`. The demos are useful for copying bits of code if you are trying to do something similar. To get an idea of what sorts of things you can do, you might want to try `AlphaImageDemoOSX`, `DotDemo`, `DrawSomeTextOSX`, `MouseTraceDemoOSX`, `DriftDemoOSX`, `MovieDemoOSX`, `PlayMoviesDemoOSX`, `PlayDualMoviesDemoOSX`, `DectectionRTInVideoDemoOSX`, `LoadMovieIntoTexturesDemoOSX`. To run these, just type the name of the program at the command line. You can press the escape key to exit from most of these programs. Note that on Macs, the movie demos may give you warnings or instructions to run a script to allow PTB to change the priority level under which it runs (higher priority can demand a higher percentage of the CPU time). You will need administrator access to your computer to run the setup script, but this is something you will need to do at some point.

If you want to look at the code for these or any other programs, use the `open` command:

```
>>open DotDemo
```

You can also tell where any function or program is located on your computer using the `which` function:

```
>>which DotDemo
```

Opening a Screen window

The core of the Psychophysics Toolbox is a function named `Screen`. Read its help file. The `Screen` function is a portal for a number of subfunctions. Type `screen` for a list.

When calling most of the screen functions, you will need to supply a screen number or “window pointer”. On Macs, which can support multiple screens, the screen number is 0 for the main screen, 1 for your second monitor, etc. PCs generally only support one screen (this may be changing in the PTB). There is a function that lists the available screens:

```
>>Screen('Screens')
```

Incidently, if you have multiple displays and are concerned about precise stimulus timing, you should run the screens in mirrored mode (so that both monitors show the same screen).

Each time you open a screen with the `Screen('Openwindow')` command, the function returns a window pointer. You should use this pointer to refer subsequently to the screen, for drawing and other commands.

First, read the help file for the `Screen('Openwindow')` function:

```
>>Screen('Openwindow?')
```

All of the help files for the `Screen` functions are accessed in this manner (adding a question mark to the end of the subfunction name).

Then, open a window:

```
>>w=Screen('Openwindow',0)
```

You can see that the function works by opening up a new screen window that takes over your desktop. If there is a bug in your program that causes the program to stop in the middle of the execution, of if you have otherwise not properly closed the screen, you may find that you seem to have lost control of your computer. But have no fear, everything is still running normally underneath the screen, you just cannot see it. One of the most annoying things about Matlab and the PTB is that there is no easy solution to this. What you need to do is to blindly type `clear Screen` to clear the `Screen` function from memory. `clear mex` will also work (this clears all of the mex functions, not just `Screen`). If this doesn't work, type control-C first (to interrupt the Matlab program that may be running) first. On the PC, you can try control-alt-delete and select the window manager tab. This should reduce the `Screen` window.

Another way to close the screen is to issue a close command.

```
>>Screen('Close?')
```

To close a particular screen, use `Screen('Close',w)`, where `w` is the window pointer returned by the `Screen('Openwindow')` function, or the window number (`0` for the main screen). You can also close all of the screens:

```
>>Screen('CloseAll?')
```

Open a screen and then close it using the three methods (`clear mex` and the two close commands). At the end of any script or function you write using the `Screen` function, make sure that you include one of the two closing functions to return control of the screen to the user.

Since blindly typing clear mex often is annoying one thing you can do is to include some error handling in the programs you will write. From here on, you should enter the various scripts in these notes into a file, using the Matlab editor. You can save the script as a `.m` file, and then you can execute the script.

To do error handling, you will use the `try catch` block design. It looks like this:

```
try
     w=Screen('Openwindow',0);
     % do stuff here – lines beginning with %'s are
     % comments and are not executed
     Screen('Close',w)
catch
     Screen('CloseAll')
     rethrow(lasterror)
end
```

All of the programs you write should look like this. The way this works is that if there is any sort of programming error in the code between the `try` and `catch` statements, rather than crashing the program and leaving you with a blank screen that you have to clear, the statements after the `catch` line will execute. In this case we automatically clear the screen and report the error.

Save the above program into a script file and run it (by clicking on the run icon, or by typing the name of the script that you save, assuming it is in your path). It will open your screen and immediately close it. It will take several seconds to open the screen. The `Screen` function performs various tests on the video hardware. If you are confident that a script that you write works fine on the computer you are using, you can skip these tests and save a few seconds by inserting the line `Screen('Preference','SkipSyncTests',1)` at the top of your program.

Before going on, we need to discuss the spatial and color coordinate systems used in the PTB.

Rects

A `rect` in the Psychophysics Toolbox is a set of rectangular coordinates `[x1 y1 x2 y2]` specifying the upper left (x_1, y_1) and lower right (x_2, y_2) coordinates of a rectangle. The origin (0, 0) of the screen is in the upper left, so y increases from the top to the bottom of the screen, and x increases from left to right. When copying parts of windows, displaying images, and so on, you will always give the position on the screen in terms of a `rect`.

When opening a screen, you can also get the `rect` of the screen you open (which must fill the entire screen for PCs but can be smaller for Macs) by requesting an additional argument from Screen openwindow:

```
[w, rect] = Screen('OpenWindow',0);
```

Then, you could compute the coordinates of the center of the screen:

```
xcenter = rect(3)/2
ycenter = rect(4)/2
```

It is useful to know the coordinates of the center of the screen since you will often be drawing stimuli relative to this point. Write a script (insert these lines into the `try catch` block above) that returns the `rect` for the screen as well as the center *x* and *y* coordinates. Make sure you understand the values in the `rect` vector.

Color

There are two main color modes that you will probably use, 8-bit color and 32-bit color. 8-bit color means that the color information for each pixel is stored in an 8-bit (i.e. one byte) number, that is, a number ranging from 0 to 255. 32-bit color is actually usually uses only 24 bits (i.e. three bytes), one 8-bit number for the red, green and blue color channels. These numbers, each ranging between 0 and 255, together form an RGB triplet.

The numbers relate to the voltages sent from your video card to your display, and are usually related to the luminance of that color channel by a function known as a gamma function. That is, the luminance is not linear with the voltage. Although this is beyond the scope of this course, when performing psychophysical experiments, you often may be concerned with the exact luminances and contrasts of your stimuli, and you therefore might be interested to learn more about this. The Psychophysics Toolbox webpage lists a number of good resources on this topic. See, e.g. http://psychtoolbox.org/tutorial.html.

Generally, you will probably work in a 32-bit color mode, although you may choose to use 8-bit color at some point for simplicity, and you should definitely understand how it works. Using 8-bit color does not mean you are limited to a choice of only 256 possible color values. Rather, you can only display 256 different colors on the screen at any one time, chosen from a larger color palette. You still have a choice of the full range of RGB values that you could use with 32-bit color. For example, if you are going to display only gray-scale images, both 32-bit and 8-bit color can be used to display the same 256 gray levels. Some video cards can display more than 8-bits per channel—10-bit cards are becoming more common. These permit a finer gradation in color brightness.

The way that 8-bit color works is through a color lookup table (CLUT). The CLUT is a 256×3 matrix, each row containing an RGB triplet. When you display a color to the screen in 8-bit mode, the number you are writing is a reference to the CLUT entry. Drawing something with color 0 will draw it using the first entry of the CLUT, and so on.

Whenever a color arguments is required in the PTB, you will be sending one of four things:

1. `[ci]` 8-bit color index
2. `[r g b]` 32-bit RGB triplet
3. `[ci a]` 8-bit color index with alpha channel
4. `[r g b a]` 32-bit RGB triplet with alpha channel

Alpha is a transparency parameter, which is used when images overlap. More on alpha blending in the last example.

Double buffering

In general, the new PTB uses double buffering in drawing things to the screen. What this means is that there are two memory buffers in the video card for graphics that will appear on the display. The first buffer is the onscreen buffer and contains the graphics that are currently being displayed. The second buffer is an offscreen buffer. All of the graphics drawing is done to this offscreen buffer, and once all of the drawing is completed, this offscreen buffer is rapidly flipped with the onscreen buffer so that the offscreen items become visible. This way, all of the drawn items appear simultaneously and can be properly synchronized with the display's refresh rate. This becomes particularly crucial when drawing items in rapid succession, such as in an animation. More on this in two weeks when we start doing some animations.

For now, what you need to know is that the things that you try to draw to the screen will not become visible until you flip the two buffers. This is accomplished with the `Screen('Flip',w)` command. This function should be called every time you have finished drawing a group of items that you would like to appear temporally together. By default, the `Flip` command moves the offscreen buffer onscreen and then creates a blank offscreen buffer, so you have to draw everything over each time you flip buffers.

The maximum speed at which you can flip the screen buffers is determined by the refresh rate of your monitor. You can get this using `hz=Screen('FrameRate',w)`, which returns the frame rate in Hz reported by your video card. For LCD displays, this may return a nonsense value, so you need to use `frame_duration=Screen('GetFlipInterval',w)` instead, which returns the length (in seconds) of each frame. The frame duration is the inverse of the frame rate.

Try putting these frame rate commands into a program after an `OpenWindow` command (remember to use the `try catch` design, which is omitted here).

```
[w, rect]=Screen('Openwindow',0);
hz=Screen('FrameRate',w)
frame_duration=Screen('GetFlipInterval', w)
frame_rate = 1/frame_duration
Screen('Close',w)
```

Drawing shapes

Now that we have all of those preliminaries out of the way, let's start drawing some things to the screen. The `Screen` function includes a number of different drawing subfunctions, including `FillRect`, `FrameRect`, `FillOval`, `FrameOval`, all of which take a `color` and `rect` argument. Try drawing a few shapes to the screen in different colors and positions, e.g., add the following line to your script:

```
Screen('FillRect',w,[128 128 128],[300 300 400 400])
```

,Don't forget to include a `Flip` command immediately after, or you won't see anything. Also, you will want to keep the shapes visible for a few seconds before closing the screen, so add in a `WaitSecs(3);` command. You could also add in a `KbWait;` command to wait for a key press (more on keyboards and timing next week).

Calling `FillRect` with no `rect` argument defaults to the entire screen. You can use this command to change the background color, e.g.:

```
Screen('FillRect',w,[255 0 0])
```

Other drawing subfunctions include `FillPoly`, which draws a filled polygon and requires a list of (x, y) verticies rather than a `rect` argument, `FillArc` and `FrameArc`, which draw curved lines, and `DrawLine`, which draws a line between two points. For the frame drawing commands (`FrameRect`, `FrameOval`, `FrameArc`, `FramePoly`), you can also specify the width and height of the frame lines, and the type of line. Try these out.

Drawing text

You can display text with the `Screen('DrawText')` subfunction, e.g.

```
Screen('DrawText',w,text,x,y,[color])
```

Where `text` is your text string (either in a variable or within single quotes), `x` and `y` are the starting positions, and `color` is color index or vector.

You can change the font, size and style of the text with the `TextFont`, `TextSize` and `TextStyle` sunfunctions. These functions need to be called before the text is drawn, and affect only subsequent text. Often you will want to know the dimensions of the text you are drawing, for instance to center the text. You can do that using the `Screen('TextBounds')` function.

For example, if you want to center some text on the screen, relative to the `xcenter` and `ycenter` coordinates (you may have computed these above):

```
text='This text is centered and red';
width=RectWidth(Screen('TextBounds',w,text));
Screen('DrawText',w,text,xcenter-width/2,ycenter,[255 0 0])
```

Try drawing text in various fonts, sizes and colors.

Displaying images

To draw an image to the screen, there are several steps. First, the image needs to be contained within a matrix, either by generating the image, or by loading it in. Second, the matrix is converted to an object called a texture that the video card can quickly access using the `Screen('MakeTexture')` function. (This is the best way. There are other ways by there is no reason to use them.) Third, this texture needs to be placed on the screen with the `Screen('DrawTexture')` function. Here's an example.

```
xsize = 300;
ysize = 200;
pict = 256*rand(ysize,xsize,3); % random colored noise
x=100;  % drawing position relative to center
y=200;
ang = 0;    % rotation angle (degrees)
[w,rect]=Screen('OpenWindow',0);
x0 = rect(3)/2; % screen center
y0 = rect(4)/2;
destrect = [x0-xsize/2+x, y0-ysize/2+y, x0+xsize/2+x, ...
y0+ysize/2+y];
t=Screen('MakeTexture',w,pict);
Screen('DrawTexture',w,t,[],destrect,ang);
Screen('Flip',w);
KbWait; % waits for a key to be pressed
Screen('CloseAll')
```

The value in creating a texture is that they are easily resized and rotated. Try changing the rotation angle and position. You can scale the image size by making the size of the destination `rect` smaller or larger than the image size. For example, try replacing `destrect` in the above code with:

```
s = 2;
destrect = [x0-s*xsize/2+x, y0-s*ysize/2+y, ...
     x0+s*xsize/2+x, y0+s*ysize/2+y];
```

If `s` is larger than one, the image is scaled up. If it is smaller than one, it is scaled down. Note that the three dots in the `destrect` line simply mean that the line breaks and continues on the next line. In your Matlab editor, the screen is probably wide enough that you won't have to split the line.

You can also load an image from a file using Matlab's `imread` function. Replace the first three lines of the above program with:

```
pict=imread('myimage.jpg');
[ysize,xsize,ncolors]=size(pict);
```

(Use your own image file, or ask Keith for one.)

Notice the size of `pict`:

```
>>size(pict)
```

There are three entries in the third dimension, corresponding to the RGB values.

What happens if you don't pass all three RGB values, but instead only one of them?

```
t=Screen('MakeTexture',w,pict(:,:,1));
```

Generating patterns

Sometimes you may need to generate images rather than just loading them in. Any image that can be stored in a matrix can be displayed to the screen. For example, here is a sine wave.

```
[x y] = meshgrid(-8*pi:pi/25:8*pi, -8*pi:pi/25:8*pi);
pict = floor(128*(sin(x)+1));
```

You can also look at the image in a figure window (outside of the Psychophysics Toolbox screen):

```
image(pict)
colormap(gray(256))
```

The Psychophysics Toolbox demo program `GratingDemoOSX` also gives an example of the creation and display of a grating with a Gaussian envelope (so it fades along its edges).

Checkerboard patterns are easy to make, for example:

```
pict=sin(x).*sin(y);
imagesc(pict)

pict=floor(255*(sign(pict)+1)/2);
image(pict)
```

You might notice that along some edges of the checkers, there is some gray that peeks through. This is because `sign(pict)` is zero where `pict` is exactly zero. You can see that with the output of `unique(pict)`, which shows all of the unique entries in the `pict` matrix. You can fix this by adding `eps`, a small number:

```
pict=floor(255*(sign(pict+eps)+1)/2);
imagesc(pict)
```

Try inserting these images into drawing program you've written above.

Let's write a script that draws images of random sizes, positions and orientations to the screen (next page).

```
% draw multiple copies of image with random size, position
% and orientation

% make sure pict, xsize and ysize are defined using one of
% the examples from above

nimages = 100; % number of images to draw
[w,rect]=Screen('OpenWindow',0);
% the next line is needed to enable transparency
Screen('BlendFunction', w, GL_SRC_ALPHA, ...
     GL_ONE_MINUS_SRC_ALPHA);
x0 = rect(3)/2; % screen center
y0 = rect(4)/2;
maxx = rect(3);
maxy = rect(4);
alpha = 1;     % transparency for overlapping images
% vary this between 0 and 1 for more or less transparency
t=Screen('MakeTexture',w,pict);
for i=1:nimages
     % choose random scale, position and angle
     s = maxx/xsize*10^(2*rand-2);
     x = x0*(2*rand-1);
     y = y0*(2*rand-1);
     ang = 360*rand;
     destrect = [x0-s*xsize/2+x, y0-s*ysize/2+y, ...
     x0+s*xsize/2+x, y0+s*ysize/2+y];
     Screen('DrawTexture',w,t,[],destrect,ang,[],alpha);
end
Screen('Flip',w);
KbWait; % waits for a key to be pressed
Screen('CloseAll')
```