FAQ - Miscellaneous tips for improved performance and timing

Q: How can i improve the drawing performance and timing of my PTB-3 code?
A: There are many measures that one can combine in order to increase reliabilty of presentation timing and drawing performance. Many of them are not neccessary on modern hardware unless you create very demanding and complex stimuli. However, they may be useful for old hardware and it is always a good idea to write your code in a portable and efficient way.

System setup:

- On MacOS/X *do not run the Mac OS9 Classic emulation environment*! It will kill any kind of precise timing behaviour!
- During a study exit all unneeded applications, applets and other gizmos.
- Disable virus scanners and other automatic periodic processes while running your study!
- Install sufficient amounts of RAM in your machine. Use of the hard drives as virtual memory will cause non-deterministic timing if your RAM is insufficient for your application.
- Make sure that the energy saving options of your machine do not slow down the processor and other subsystems to save power or battery life. Especially important for laptops on battery power.
- It may help to disable or disconnect Bluetooth, WLAN/Wifi and network connectors and to unload CD or DVD disk drives.
- Don't run your study at excessive screen resolutions, choose something reasonable for your purpose. Higher resolutions take up more precious VRAM graphics memory and increase the time for clearing the framebuffer, drawing your stim and for post-processing operations.

Matlab:

- Try to run the study in *matlab -nojvm* mode --> Cuts down all the periodic processing done by the Matlab graphical user interface and disables the heavy-weight and ressource-hungry Java virtual machine, reduces memory load and cpu load. This may also help to resolve "out of memory" errors on older Matlab releases (caused by Java) and crashes and hangs related to Quicktime movie playback with some movie formats (caused by bugs in Matlabs Java and Apples Quicktime or bad interactions between them). This doesn't apply if you need *GetChar* and friends - they depend on Java, at least on OS-X.
- If you can't or don't want run in *matlab -nojvm* mode, try to minimize Matlabs user interface processing: Close unneccessary Matlab windows and views. E.g., the view window for the contents of your current working directory gets updated every couple of seconds when shown - expensive, can lead to additional periodic timing jitter!
- Read the documentation of Matlab about optimizing your code for efficient use of memory and speed.
- Use Matlabs mlint *help mlint* tool and its profiler *help profile*: They hint you to performance bottlenecks and other problems in your code.

Dual display operation:

Running two displays simultaneously requires more VRAM memory and computational ressources in your gfx-card, plus some additional management overhead in the graphics driver. This can cause lower graphics performance and therefore lead to missed stimulus onset deadlines when running complex stimuli on high resolution, high refresh rate displays, especially when using older mobile graphics chips with a low amount of local VRAM. To minimize these effects, try the following:

- Use dual display mode only if you need two displays for your study, e.g., stereo display, haploscopes and such. Disable the 2nd display during experiment sessions if you don't need it. Its fine to run dual display studies where PTB has full control over both displays, i.e. one onscreen window completely spanning and covering both displays, or two separate onscreen windows, each of them completely covering each one of the displays. PTB is optimized for this usage case, taking control over ressource management for such a setup.

- If you can't or don't want to disable the 2nd display, although you don't need it for your study, e.g., Laptops where internal panel can't be disabled, try the following tricks:
    - Switch the displays into mirror mode or clone mode with same resolution, color depths and refresh rate: This may or may not help reducing load, depending on the design of your graphics hardware, driver and operating system.
    - Open a 2nd onscreen window on the display which you don't need. The window has the sole purpose of granting PTB control of the unneeded display, blocking other applications and the GUI from updating it.
    - If none of both is possible or helpful, try to minimize any graphics updates on the display with your desktop: Minimize or close unneeded applications, disable the "seconds" display and indicators on the clock, calm down other little applets.

Programming:

Psychtoolbox output to the Matlab window

The amount of verbosity of PTB can be controlled via the *Screen('Preference', 'Verbosity', level);* command. Use low settings for *level* to minimize output to the Matlab window: Reduced clutter and saved time for printing. Use high (or default) settings during development of your code and during first test runs to make sure that you get informed by PTB of possible issues with your setup or code!

Settings for the *level* parameter and their meaning:

1. No output at all.
2. Only severe error messages.
3. Errors and warnings.
4. Additional information, e.g., startup output when opening an onscreen window.
5. Even more information and hints.
6. Very verbose output, mostly useful for debugging PTB itself. Usually not what you want.

Preloading of functions:

Matlab M-Files and especially MEX files do have some additional delay on first invokation (Matlab needs to find and load them, compile or link them, they need to perform some internal initialization). This can be significant, e.g., multiple hundred milliseconds. For time critical studies, use each function once before your trial loop to "preload" the function before first use.

M-Functions are faster than M-Scripts:

Add the *function* keyword to the top of your experiment script after initial testing and debugging. Matlab precompiles functions into some optimized code at load time, so they execute faster than scripts (M-Files without *function* keyword).

If the executed m-file is a script then Matlab reparses and interprets each statement again every time the script is executed and no optimizations are applied. It is as if you type commands into the matlab window, just typing very fast.

Functions, however, are precompiled and optimized by Matlab's JIT compiler at load time, and during execution, so they are faster. One problem of dynamic optimization at runtime ("hot spot optimization") is that loops in your code may execute slower during the first few iterations than in the remaining iterations, because they are optimized during runtime. Because of this, it is recommended to dry-run your code for a few trials before starting your actual experiment. This is beautiful for most applications, but can create headaches for benchmarking and deterministic timing. The other problem is the lack of documentation and control about these features. The strategies apparently change with each new matlab release, as new optimization features are added under the hood.

This JIT mechanism is what gives Matlab a speed advantage over Octave for poorly written code, whereas there's basically no difference for well written code in most cases. For maximum performance and most deterministic timing it is best to optimize your code yourself as described in Mathworks documentation, so the JIT has to do less guesswork and runtime optimization and you'll get better performance/timing from the start. And of course use ptb's timing functions wisely to immunize your code against such behaviours as much as possible.

Other M-code optimizations: Good coding practices

With some googling, quite a bit can be found on good coding practices. I have pasted a few links that seem good below.
It is suggested you start by reading MCFEX: Writing Fast MATLAB Code.
See Peter Acklam's MATLAB array manipulation tips and tricks for many techniques for writing vectorized code.
The below links seem to have some useful information and tips too (add any links that you found useful and do not simply duplicate information already introduced in other links):
Speed MATLAB by optimizing memory access
For tips on measuring performance of scripts, see MCFEX: MATLAB Performance Measurement. This file also contains a section *A Catalog of Noise*, which introduces you to warming up cold code and to sources of potential timing variability.

But don't forget, If your code works well and timing is good, don't spend hours optimizing it--the payoff for your work would be almost nil and you might even introduce unforseen bugs. In my view, this statement doesn't hold if your new to MATLAB and coding practices that work well. Experiment and learn! do take your time to optimize, following the pointers from the links above. That way you wil make good practice your own and you will learn to think in good code (vectorized code is easier on your working memory also).

Realtime scheduling:

For time critical code, use the *Priority(prio)* command to raise Matlab to realtime-priority mode. Select the *prio* value via *prio=MaxPriority(...)* to get the highest suitable priority for your needs in a portable, operating system independent way. Don't forget to disable realtime mode via *Priority(0)* at the end of your script!

Optimal Code structure:

If you experience missed frame presentation deadlines, try to structure your code inside your animation/trial loop as follows:

1. All *Screen* drawing commands, e.g., *Screen('DrawDots',...) , Screen('DrawText', ...), Screen('DrawTexture', ...).*
2. The command *Screen('DrawingFinished', win[, clearmode]);* where *clearmode* is the same optional flag that you'll optionally pass to the *Screen('Flip', win);* command. This will tell PTB that stimulus drawing for this frame is finished, so it can perform a couple of optimizations, e.g., perform potential stimulus post-processing operations immediately and in parallel to the execution of your non-graphics Matlab code.
3. All commands not related to graphics, e.g., sound output, keyboard/mouse/response box queries, Matlab code, ...
4. The *vbl=Screen('Flip', win [,when]);* command to trigger stimulus onset. Providing a *when* timestamp is recommended. It allows PTB to further optimize drawing when it knows, *when* exactly the stimulus should be shown. It also allows for higher reliability of the built-in frame skip detector and it allows for very robust stimulus onset timing. On most hardware and operating systems, the returned *vbl* timestamp is a highly accurate and robust timestamp of the time your stimulus was flipped onto the

display, unaffected by all the timing noise that is invariably present in most operating systems.

While use of *Screen('Flip', ...)* with the *vbl* and *when* arguments is nearly always recommended for studies where you care about frame accurate timing, use of the *Screen('DrawingFinished',...)* command is optional and not needed for many studies. Its worth a try if you have very complex and demanding stimuli, tight deadlines at high framerates or old and slow hardware: It doesn't hurt to use it, but it doesn't help in all cases either.

DriftWaitDemo.m in the *PsychDemos* folder of PTB is a good example of how to provide timed onset of stimuli.

Reduced VRAM usage on graphics hardware with very small VRAM

Try if adding a *Screen('Preference', 'ConserveVRAM', x);* with x=1 or 2 or 3 or 4 to the top of your script makes things better or worse. Reduced VRAM pressure, but higher load for system RAM and bus. Under these settings, PTB doesn't cache textures in VRAM anymore but only keeps them in system memory.

Speeding up specific PTB drawing commands

- *Screen('OpenWindow')*: Full scene smoothing/anti-aliasing via settings of the optional *multisample* parameter of > 0 is an expensive operation, both in terms of memory consumption and drawing speed! Higher *multisample* levels are more expensive. If you only need to draw anti-aliased textures, dots and lines, use the smoothing modes of *Screen('DrawDots') and Screen('DrawLines')* and the bilinear texture filtering of *Screen('DrawTexture')* instead. Only use full scene anti-aliasing for more complex or combined stimuli.

- *Screen('Flip')*: If you have an animation loop where your stimulus is constantly and fully overdrawn by following stimuli and you don't use alpha-blending, then you can skip clearing of the framebuffer after each *Screen('Flip', win, when, clearmode, ...)* command by setting the optional *clearmode* flag to a value of 2. This will prevent PTB from clearing the framebuffer to background color after each flip. That way you can save 1-2 milliseconds per frame on old and slow graphics hardware, maybe more on onboard graphics chips. On modern hardware, the gain will be only fractions of a millisecond, if measurable at all. *clearmode=1* is the slowest mode of operation, unless you have the Psychtoolbox imaging pipeline enabled, there it is for free. This clearmode is only beneficial for incremental updating of very complex stimuli, where saved redraw time may outweight the increased cost of this mode.

- *Screen('OpenOffscreenWindow')*: Use only if you need to draw to an offscreen surface for precomputation of your stimuli. Most stimuli can be drawn on-the-fly thanks to modern fast graphics hardware and the double-buffered drawing model of PTB-3. Matlab image matrices are best drawn via the *Screen('MakeTexture') and Screen('DrawTexture')* commands. Use Offscreen windows only if you need to draw into them with Screen 2D drawing commands or MOGL 3D drawing commands. Try to restrict the size of an Offscreen window to the minimum needed size via the optional *rect* parameter - Most stimuli don't cover the whole screen after all. Switching between drawing into different Offscreen windows is expensive, unless you use the PTB imaging pipeline. Therefore try to group drawing commands which draw into the same Offscreen window, thereby minimizing the number of switches between Offscreen windows.

- *Screen('DrawText')*: This is fast and low-quality on MS-Windows and beautiful but slow on OS/X. If you need to draw the same texts repeatedly in very timing critical loops, you may want to draw the text into an offscreen window, thereby creating a text slide for later presentation via *Screen('DrawTexture')*. Switching text properties like size, style and especially font can be an expensive operation on some operating systems. Group text drawing commands which share common text properties together, thereby minimizing changes of settings.

- *Screen('PutImage')*: This is a slow command. Use the combo of *Screen('MakeTexture') and Screen('DrawTexture')* instead if possible. See *DriftDemo.m* for a simple example.

- *Screen('DrawTexture')*: If you draw textures without any scaling, subpixel scrolling or subpixel shifts (*srcRect* and *dstRect* of

same size and positioned at integral pixel positions) and without rotation (standard upright drawing), you can disable bilinear texture filtering by explicitely setting the optional flag *filtermode* of *Screen('DrawTexture',.....,filtermode)* to zero. This will reduce the amount of data being processed by a factor of four. On modern graphics hardware, speed-gain may be minimal, but older hardware or onboard graphics chips may benefit. N.b.: If you didn't know about subpixel scrolling, play around with *DriftDemo2.m or DriftDemo3.m* and their parameters. They allow for drifting gratings that drift at a speed below the motion threshold, without any need to precompute grating movies or textures.

- *Screen('FillPoly')*: Filling convex polygons is faster than filling concave or self intersecting polygons. This will only matter for large polygons with many points.

- *Screen('DrawDots') and Screen('DrawLines')*: Drawing dots or lines with a single common color and size is the fastest. Drawing dots and lines with a color that is individual to each specific dot or line is slower. Drawing dots or lines of individual diameter or width for each dot or line is significantly slower. Drawing dots or lines of big diameter or width or with point- or line smoothing enabled may be slower on old graphics hardware or onboard graphics chips, but won't matter very much on modern hardware.

TESTS:

The following scripts may be useful to assess display timing on your setup:

- The *VBLSyncTest* script is a good way to assess and learn about display timing. It has many mandatory parameters so read *help VBLSyncTest* carefully.
- *CheckFrameTiming* is another test script.
- *PerceptualVBLSyncTest* is a visual test to assess proper synchronization of stimulus onset to the monitor refresh cycle.

...to be continued, stay tuned...