MGL

Mgl is a suite of mex/m files for displaying psychophysics stimuli in Matlab. Runs on Mac OS X (G4/5 and Intel) and Linux. Version 1.10

A quick overview

mgl is a set of matlab functions for dispalying full screen visual stimuli from matlab. It is based on OpenGL functions, but abstracts these into more simple functions that can be used to code various kinds of visual stimuli. It can be used on both Linux and Mac OS X systems. Stimuli can be displayed full screen or in a window (helpful for debugging on a system that only has one display). With a single command that specifies the distance to and size of a monitor, the coordinate system can be specified in degrees of visual angle, thus obviating the need to explicitly convert from the natural coordinate frame of psychophysics experiments into pixels. The best way to see whether it will be useful to you is to try out the mglTest programs (see 1.2 below) and also the sample experiment testExperiment. A basic “hello world” program can be written in four lines:

Open the screen, 0 specifies to draw in a window. 1 would be full screen in the main display, 2 would be full screen in a secondary display, etc...

```
>> mglOpen(0);
```

Select the coordinate frame for drawing (e.g. for a monitor 57 cm away, which has width and height of 16 and 12 cm).

```
>> mglVisualAngleCoordinates(57,[16 12]);
```

draw the text in the center (i.e. at 0,0)

```
>> mglTextDraw('Hello World!',[0 0]);
```

The above is drawn on the back-buffer of the double-buffered display so to make it show up you flush the display (this function will wait till the end of the screen refresh)

```
>> mglFlush;
```

To close an open screen:

```
>> mglClose;
```

What is in the mgl distribution

- mgl/mgllib: The main distribution that has all functions for displaying to the screen.
- mgl/task: A set of higher level routines that set up a structure for running tasks and trials. Relies on functions in mgl/mgllib. You do no need to use any of these functions if you just want to use this library for drawing to the screen.

- mgl/utils: Various utility functions.

GNU General Public License

These programs are free to distribute under the GNU General Public License. See the file mgl/COPYING for details.

Recompiling mgl

For G4/5 based macs you should not need to recompile the distribution as it comes with precompiled .mexmac files.

However, if you want to use the code on linux or if when you first start using the functions they do not work or cause an obvious error like a segmentation fault, then you may want to recompile. This requires mex to be setup properly on your machine. At a minimum you will need to have the apple developer tools installed on your machine (XCode) http://developer.apple.com/tools/ [http://developer.apple.com/tools/]: The command to recompile is:

» mglMake(1);

If all else fails, how to get back control over the display?

If you can't do mglClose, you can always press:

option-open apple-esc

this will quit your matlab session as well.

Can I get access to all OpenGL functions?

We have only exposed parts of the OpenGL functionality. If you need to dig deeper to code your stimulus, consider writing your own mex file. This will allow you to use the full functionality of the OpenGL library. To do this, you could start by modifying one of our mex functions (e.g. mglClearScreen.c) and add your own GL code to do what you want and compile.

Printing the wiki help pages

You can print out all the wiki help pages at once, by using this link.

MGL Download

External Users

Click on the following link to receive a tar/zip file of the current stable version:

mgl.tar.gz [http://www.cns.nyu.edu/~justin/mgl.tar.gz]

After downloading you can just click on the tar file and it should expand itself or from a command line do:

```
gunzip mgl.tar.gz
tar xfv mgl.tar
```

to extract the files.

NYU Users

mgl is available from the following directory. This will always be the latest stable version.

```
/share/wotan/heegerlab/mgl
```

NYU Users who wish to make changes to the code

You can use CVS to checkout a version that you can modify yourself.

```
> cvs -d ~justin/src checkout mgl
```

If you want to make changes only to the stable version:

```
> cvs -d ~justin/src checkout -r v1_10 mgl
```

Note that changes to the stable version should only be made to fix bugs. If you make a change to the stable version, make sure to make the corresponding change to the development version.

You may also access the code repository via the web by adding the following to your .cshrc.mine file

1. for csh or tcsh shell:

   ```
   setenv CVSROOT :ext:justin@cns.nyu.edu:/users/justin/src
   setenv CVS_RSH ssh
   ```

2. for bash shell:

   ```
   export CVSROOT=:ext:justin@cns.nyu.edu:/users/justin/src
   export CVS_RSH=ssh
   ```

You will need to change justin@cns.nyu.edu to your own login.

Also, you will need to have the cvs binary somewhere in your path on your account. You can copy the cvs binary from

```
~justin/bin/cvs
```

Latest MGL version

To make sure you have the latest mgl version cd to your local folder of mgl and type:

```
cvs update
```

Initial setup

Simply add the mgl directory to your path, and you are ready to go.

```
>> addpath(genpath('MYPATH/mgl'));
```

where MYPATH should be replaced by the path to your version of mgl.

You can see what functions are available by doing (in matlab):

```
>> help mgl
```

There are a bunch of test programs (names start with mglTest) that you can use to test the distribution and see how things are done.

If you want to run under linux, then you will need to compile the distribution, using mglMake (see below).

You may also need to recompile the distribution for your system if you are running an older version of matlab (we run Matlab version >= 7.3 on Mac OS >= 10.4.8). We have found that mex files created on Matlab 7.3 do not run on matlab 14.1 for instance (if you run -nojvm you will see that it complains that it cannot find a dynamic link library for the mx functions-if you run with the matlab desktop it will just crash the system). If this happens to you simply recompile and you should be good to go.

Rebuilding mex files

We (Apple developers) run the latest Mac OS (10.5.6 as of this writing) with the latest version of Matlab (7.6 as of this writing) or (Linux developers) Ubuntu 64-bit (Gutsy) and 32-bit (Feisty) with Matlab 7.4 and the binaries are created to run on these systems. As noted above, some older versions (notably Matlab 14.1) are not able to use these mex files and crash when you try to run mglOpen. If this happens, then all you need to do is rebuild the distribution using mglMake(1).

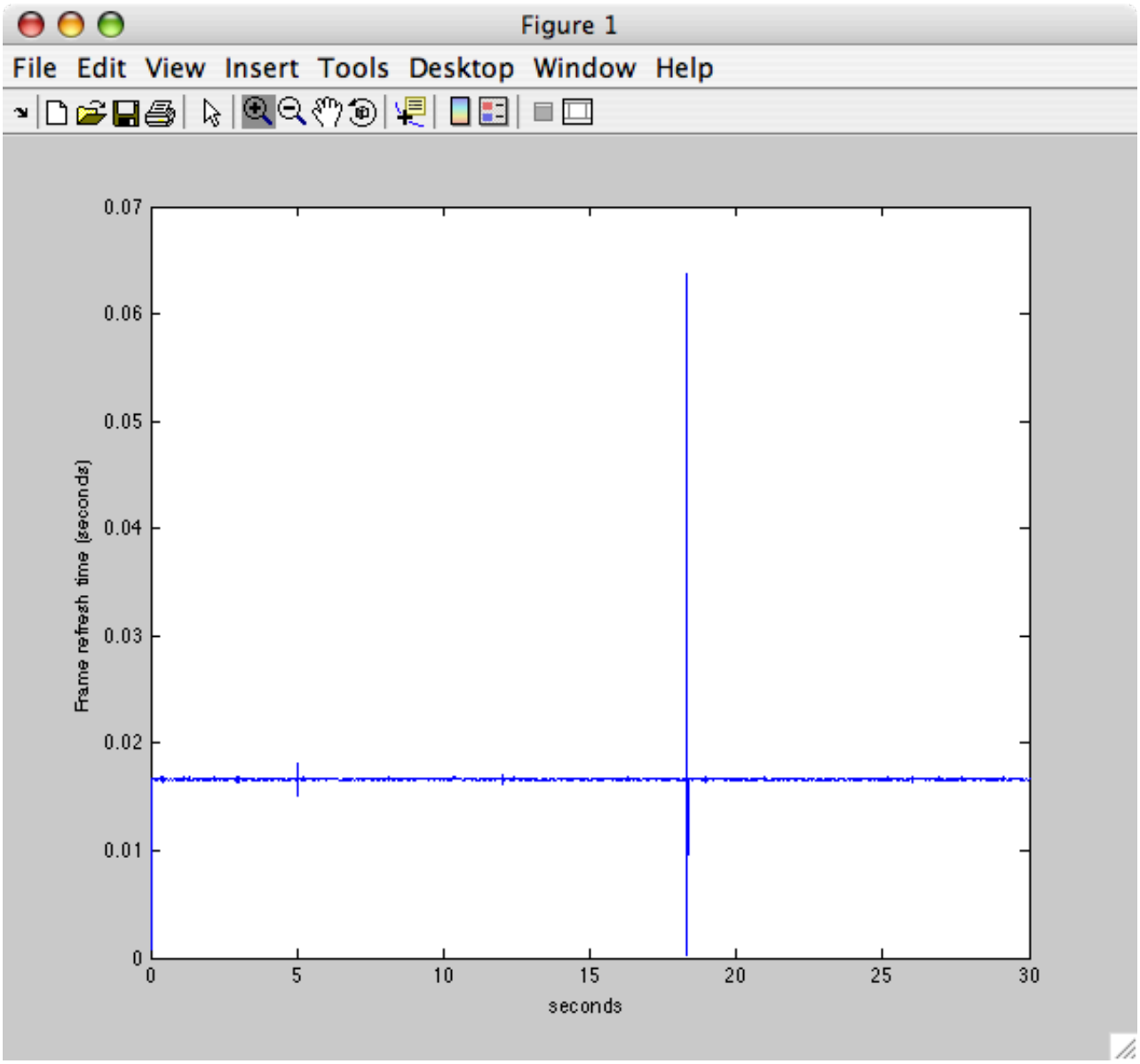
License manager timing glitch

The Matlab license manager checks every 30 seconds for the license. This can cause there to be an apparent frame glitch in your stimulus code, especially if you are using a network license (on our machines it can take ~200 ms to check for the license). The only known workaround to this is to run on a machine that has a local copy of the license. You can check this for yourself by seeing how long it takes to do screen refreshes:

```
mglOpen;
global MGL;
checkTime = 30*MGL.frameRate;
```

```
timeTaken = zeros(1,checkTime);
mglFlush;
for i = 1:checkTime
  flushStart = mglGetSecs;
  mglFlush;
  timeTaken(i) = mglGetSecs(flushStart);
end
mglClose;
plot((1:checkTime)/frameRate,timeTaken);
zoom on;xlabel('seconds');ylabel('Frame refresh time (seconds)');
```

If you have the same problem, you should see one large spike in the time course like this:



This one shows it taking about 65 ms. Note that you may see small deviations in which one frame takes longer and then the following frame takes shorter than the mean. These are normal flucations persumably due to multi-tasking and other events that are intermittently taking up time. As long as these are shorter than a frame refresh interval minus the time it takes you to process the stimuli for your display, you will not drop any frames. Note that in the above code, if you change mglFlush to any other command, such as WaitSecs(1/frameRate);, you will still see the big spike for the license manager check-confirming that this has nothing to do with drawing to the screen.

Function not supported on Linux

Not all functions are currently supported on the Linux platform. The list of funcitons not supported yet are:

- mglText
- mglTextDraw
- mglPlaySound
- mglInstallSound
- mglDescribeDisplays
- mglSwitchDisplay

If you want to use text under the linux operating system, you can use mglStrokeText.

Here is a more recent update from Jonas about the Linux version:

I am in the process of upgrading the Linux version of MGL to run under Ubuntu (64-bit and 32-bit) with NVIDIA and ATI graphics cards. Although the upgrade is still incomplete, most functions work equally well under Linux at this stage. Some differences that will remain between the platforms are listed below.

- no support for font-based text - this needs to be upgraded, I started looking into using FreeType for this which is widely available and would be easy to implement in the same texture-based way that the Mac code relies on. Care must be taken to ensure that the code is maximally portable across platforms, so it may be that some Mac-specific idiosyncracies need to be modified
- note that the stroke text works perfectly under Linux, so unless you are very enamoured with a specific font this is a perfectly usable workaround (though some symbols, eg %, are missing currently)
- some differences in the way you specify special keys, but this is generally to Linux' advantage - eg you can use ESC, BACKSPACE etc as names (relies on the XKeySymDef.h or sth like that)
- timing is in general more accurate on Linux, since the clock rate is much higher on modern systems (100-500Hz vs 60Hz on the Mac)
- no parallel port interface yet so you can't use Justin's code for calibration
- no sound - need to decide on a standard to use that is most widely available
- the syncing with OpenGL is idiosyncratic and depends on the graphics card. I have implemented this to use both the SGI video sync extension and the environment variable (both NVIDIA and ATI provide this option). The former is not supported by all OpenGL cards (though most modern ones) and can interact with the environment variable option, so I will make the latter the default, with an option to use the SGI extension when compiling only.

For the time being, only NVIDIA and ATI cards will be supported (because I only have access to those two machines).

- setting screen size and resolution requires the XRandR extension, which is supported on recent X distributions (Xorg 7.0 and later). Older X servers (eg Apple's X11) won't work.
- you need to reuse the X display or you run into memory problems, and the code for doing this needs to be checked for consistency. This is similar to the Mac window situation and relates to the uneasy relationship between Matlab and X. When you open a window, the MGL global variable will contain a window and display pointer that is used on subsequent calls; care must be taken not to clear the MGL variable between calls (once you do so, running MGL is likely to crash Matlab, even if you run clear all, which correctly closes the

display).

Functions not supported on Mac

- mglSetRandR allows you to set and get resolution and refresh rate from the command line. There is no support for Mac at present; use the Display Manager.

Opening in a window with matlab desktop (Mac only)

This issue had been resolved, but if you find you are still having problems, here is what the issue was and how to resolve it:

On Mac OS X there seems to be some interaction with having mutliple threads in the workspace that causes working within a window (i.e. mglOpen(0) as opposed to fullscreen) to be unstable. The workaround for now is not to close the window once it is opened. This seems to work fairly well. When one is completely finished working with the window, one can call mglPrivateClose to close the window. But after that, calling mglOpen(0) is likely to crash.

mglOpen(0) works fine if running matlab -nojvm or -nodesktop.

Precise timing of key press events

On Mac OS X if you want to get key press events, you can only get them with a resolution of 1/60 second. If you are willing to sit in a loop testing the keyboard, you can use mglGetKeys and get acceptable time resolution (your time resolution will depend on how fast you poll the keyboard status). On Linux, the time resolution depends on the kernel's hard-coded HZ setting, which is usually between 100-500; the value can be retrieved (at least on Ubuntu) by typing cat /boot/config-`uname -r` | grep '^CONFIG_HZ=' The time resolution is 1/HZ sec; so on the current developer system, with HZ=250, key presses can be timed with a resolution of 1/250=4msec.

64bit Matlab on Mac OS X

Apple has given up support for Carbon based GUIs for 64 bit applications, so internal functions have been rewritten to use the Cocoa (objective-c) based interface. Everything is working and there are some improvements.

- **Events** The functions mglGetKeyEvent, mglGetMouseEvent now return timestamps that have nanosecond precision (and not 1/60 second precision as before).
- **mglMovie** You can now display Quicktime movies.
- **Transparent windows** For mglOpen(0) you can display a transparent window by passing in a value greater than 0 and less than 1, e.g. mglOpen(0.7);

Internal notes

Functions affected:

- mglPrivateOpen: working.
- mglPrivateClose: working.
- mglGetMouse: working.

- mglGetKeyEvent: working. nanosecond time stamps. Remember to move old mex files.
- mglGetMouseEvent: working. nanosecond time stamps. Remember to move old mex files.
- mglCharToKeycode: working. update wiki page.
- mglKeycodeToChar: working.
- mglPlaySound: working. Check deallocation. Remember to remove old mglPlaySound and add mglPrivatePlaySound
- mglInstallSound: working.
- mglPrivateDescribeDisplays: working except for gammaTableWidth/Length
- mglPrivateSwitchDisplay: working.
- mglPrivateMoveWindow: working.
- mglMovie: working. add mglMovie.m and mglPrivateMovie.m

List of functions to be checked with Linux:

- mglGetMouseEvent
- mglGetKeyEvent
- mglInstallSound
- mglPlaySound
- mglPrivateOpen
- mglPrivateClose
- mglSetGammaTable
- mglGetGammaTable
- mglCharToKeycode (add support for returning multiples)
- mglKeycodeToChar
- mglMovie/mglPrivateMovie

Main screen functions

mglOpen: Opens the screen

usage: mglOpen(whichScreen, <screenWidth>, <screenHeight>, <frameRate>, <bitDepth>)
purpose: Opens an openGL window

argument	value
whichScreen	0=Window, 1=primary screen, 2=secondary screen, etc
<screenWidth>	[width] in pixels
<screenHeight>	[height] in pixels
<frameRate>	[frameRate] in hertz
<bitDepth>	[bitDepth] this is usually 32 bits

Open last monitor in list with current window settings

```
mglOpen
```

Open with resolution 800×600 60Hz 32bit fullscreen

```
mglOpen(1,800,600,60,32);
```

Open in a window

```
mglOpen(0);
```

Note that mglOpen hides the mouse cursor as a default. If you want to show the mouse cursor, you should call mglDisplayCursor after opening the screen.

mglFlush: Flips front and back buffer

purpose: swap front and back buffer (waits for one frame tick)
usage: mglFlush
N.B. mglFlush has to be called after each operation that involves drawing or erasing from the currently open window

mglClose: Closes the screen

purpose: close OpenGL screen
usage: mglClose

Other screen functions

mglSwitchDisplay: Switch between multiple monitors

usage: mglSwitchDisplay(<displayID>)
purpose: If you are using multiple monitors to display stimuli (like for a dichoptic presentation), you can open up multiple displays using this function.

argument	value
<displayID>	which display to switch to

You can use this to open up two separate screens and control them independently. For example, say you have two monitors 1 and 2. You open the first in the usual way:

```
mglOpen(1)
```

Then you switch monitors so that you can open up the other one

```
mglSwitchDisplay
mglOpen(2)
```

Now if you want draw to the first display, you can do

```
mglSwitchDisplay(1)
mglClearScreen(0.5);
mglFlush;
```

Similarly, to draw to the second display

```
mglSwitchDisplay(2);
mglClearScreen(1);
mglFlush;
```

You can check the status of all open displays with:

```
mglSwitchDisplay(-2);
```

If you want to close all displays at once, you can do:

```
mglSwitchDisplay(-1);
```

mglMoveWindow: Moves windows created by mglOpen(0)

usage: mglMoveWindow(leftPos,topPos)
purpose: Moves a window created by mglOpen(0)

argument	value
leftPos	Left position of where the window will be moved to
topPos	Top position of where the window will be moved to

```
mglOpen(0);
mglMoveWindow(100,100);
```

mglDescribeDisplays: Get information about your monitor and computer system

usage: [displayInfo computerInfo] = mglDescribeDisplays()
purpose: Gets information about your displays (as an array of structs with values for each monitor). As well as a single struct with info about your computer.

mglFrameGrab: Frame grab to a matlab matrix

usage: mglFrameGrab(<frameRect>)
purpose: Does a frame grab of the current mgl screen and returns it as a matrix of dimensions widthxheightx3

argument	value
frameRect	Optional argument that is a 1×4 array specifying a rectangular part of the frame to grab in the format [x y width height]

```
mglOpen();
mglScreenCoordinates;
```

```
mglClearScreen([0 0 0]);
global MGL;
mglPoints2(MGL.screenWidth*rand(5000,1),MGL.screenHeight*rand(5000,1));
mglPolygon([0 0 MGL.screenWidth MGL.screenWidth],[MGL.screenHeight/3 MGL.screenHeight*2/3 MGL.screenHe
 mglTextSet('Helvetica',32,[1 1 1]);
mglTextDraw('Frame Grab',[MGL.screenWidth/2 MGL.screenHeight/2]);
frame = mglFrameGrab;
mglFlush
imagesc(mean(frame,3)');colormap('gray')
```

Functions to adjust the coordinate frame

mglVisualAngleCoordinates: Visual angle coordinates

purpose: Sets view transformation to correspond to visual angles (in degrees) given size and distance of display. Display must be open and have valid width and height (defined in MGL variable)
usage: mglVisualAngleCoordinates(physicalDistance,physicalSize);

argument	value
physicalDistance	[distance] in cm
physicalSize	[width height] in cm

```
mglOpen
mglVisualAngleCoordinates(57,[16 12]);
```

mglScreenCoordinates: Pixel coordinate frame

purpose: Set coordinate frame so that it is in pixels with 0,0 in the top left hand corrner
usage: mglScreenCoordinates()

mglTransform: Low-level function to adjust transforms

purpose: applies view transformations
usage: mglTransform(whichMatrix, whichTransform, [whichParameters])

argument	value
whichMatrix	'GL_MODELVIEW', 'GL_PROJECTION', or 'GL_TEXTURE'
whichTransform	'glRotate', 'glTranslate', 'glScale','glMultMatrix', 'glFrustum', 'glOrtho','glLoadMatrix', 'glLoadIdentity', 'glPushMatrix','glPopMatrix', 'glDepthRange', or 'glViewport'
whichParameters	function-specific; see OpenGL documentation

You can also specifiy one of GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE and a return variable current matrix values. If two outputs are specified, the result of the computation will be returned.

This function is usually not called directly, but called by mglVisualAngleCoordinates or mglScreenCoordinates to set the transforms

mglHFlip: Horizontally flip coordinates

purpose: flips coordinate frame horizontally, useful for when the display is viewed through a mirror
usage: mglHFlip()

```
mglOpen
mglVisualAngleCoordinates(57,[16 12]);
mglHFlip
mglTextSet('Helvetica',32,1,0,0,0,0,0,0,0);
mglTextDraw('Mirror reversed',[0 0]);
mglFlush;
```

mglVFlip: Vertically flip coordinates

purpose: flips coordinate frame vertically
usage: mglVFlip

```
mglOpen
mglVisualAngleCoordinates(57,[16 12]);
mglVFlip
mglTextSet('Helvetica',32,[1 1 1],0,0,0,0,0,0,0);
mglTextDraw('Vertically flipped',[0 0]);
mglFlush;
```

Texture functions used for displaying images

mglCreateTexture: Create a texture from a matrix

purpose: Create a texture for display on the screen with mglBltTexture image can either be grayscale nxm, color nxmx3 or color+alpha nxmx4
usage: texture = mglCreateTexture(image,axis)

argument	value
image	nxm matrix of grayscale values from 0 to 255, or nxmx3 matrix of RGB values or nxmx4 of RGBA values
axis	'xy' rows are x and columns are y dimension (default–matlab oriented matrix, i.e. will give you the same results as using imagesc), 'yx' rows are y and columns are x dimension
OUTPUT: texture	a texture structure that can be drawn to the screen using mglBltTexture

```
mglOpen;
mglClearScreen
mglScreenCoordinates
texture = mglCreateTexture(round(rand(100,100)*255));
mglBltTexture(texture,[0 0]);
mglFlush;
```

mglBltTexture: Draw the texture to the screen

purpose: Draw a texture to the screen in desired position.
usage: mglBltTexture(texture,position,<hAlignment>,<vAlignment>,<rotation>)

argument	value
texture	A texture structure created by mglCreateTexture or mglText.
position	Either a 2-vector [xpos ypos] or 4-vector [xpos ypos width height]. units are degrees.
hAlignment	-1 = left, 0 = center, 1 = right (defaults to center)
vAlignment	-1 = top, 0 = center, 1 = bottom (defaults to center)
rotation	rotation in degrees, defaults to 0

To display several textures at once, texture can be an array of n textures, position is nx2, or nx4 and hAlignment, vAlignment and rotation are either a single value or an array of n.

multiple textures:

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
image = rand(200,200)*255;
imageTex = mglCreateTexture(image);
mglBltTexture([imageTex imageTex],[-3 0;3 0],0,0,[-15 15]);
mglFlush;
```

single textures

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
image = rand(200,200)*255;
imageTex = mglCreateTexture(image);
mglBltTexture(imageTex,[0 0]);
mglFlush;
```

mglDeleteTexture: Delete a texture

purpose: Deletes a texture. This will free up memory for textures that will not be drawn again. Note that when you call mglClose texture memory is freed up. You only need to call this if you are running out of memory and have textures that you do not need to use anymore.
usage: mglDeleteTexture(tex)

argument	value
tex	The texture created by mglCreateTexture that you want to delete

```
mglOpen;
mglClearScreen
mglScreenCoordinates
texture = mglCreateTexture(round(rand(100,100)*255));
mglBltTexture(texture,[0 0]);
mglFlush;
mglDeleteTexture(texture);
```

Drawing text

mglTextSet: Set parameters for drawing text

purpose: Set text properties for mglText
usage:
mglTextSet(fontName,<fontSize>,<fontColor>,<fontVFlip>,<fontHFlip>,<fontRotation>,<fontBold>,<fontItalic>

argument	value
fontName	'fontName' (defaults to 'Times Roman')
fontSize	font point size (defaults to 36)
fontColor	[r g b] where r, g and b are values between 0 and 1 (defaults to [1 1 1])
fontVFlip	0 = no vertical flip, 1 = vertical flip (defaults to 0)
fontHFlip	0 = no horizontal flip, 1 = horizontal flip (defaults to 0)
fontRotation	rotation in degrees (defaults to 0)
fontBold	0 for normal, 1 for **bold** (defaults to 0)
fontItalic	0 for normal, 1 for *italic* (defaults to 0)
fontUnderline	0 for normal, 1 for *underline* (defaults to 0)
fontStrikethrough	0 for normal, 1 for strikethrough (defaults to 0)

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglTextSet('Helvetica',32,[0 0.5 1 1],0,0,0,0,0,0,0);
mglTextDraw('Hello There',[0 0]);
mglFlush;
```

mglText: Create a texture from a string

purpose: Creates a texture from a string.
usage: tex = mglText('string')

argument	value
string	The string you want to draw

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglTextSet('Helvetica',32,[0 0.5 1 1],0,0,0,0,0,0,0);
thisText = mglText('hello')
mglBltTexture(thisText,[0 0],'left','top');
mglFlush;
```

Normally you will only set one output argument which is a texture usable by mglBltTexture. But if you have two output arguments

```
[tex texMatrix] = mglText('hello');
```

texMatrix will contain a 2D matlab array that has a rendering of the text (i.e. it will have values from 0-255 that

represent the string). You can modify this matrix as you want and then use mglCreateTexture to create it into a texture that can be displayed by mglBltTexture

mglTextDraw: Draws text to screen (simple but slow)

purpose: wrapper around mglText and mglBltTexture to draw some text on the screen. If you need to draw text more quickly, you will have to pre-make the text textures with mglText and then use mglBltTexture when you want it. Otherwise, for non time-critical things this functions should be used.
usage: mglTextDraw(str,pos,<hAlignment>,<vAlignment>)

argument	value
str	desired string
pos	[x y] position on screen
hAlignment	-1 = left, 0 = center, 1 = right (defaults to center)
vAlignment	-1 = top, 0 = center, 1 = bottom (defaults to center)

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglTextSet('Helvetica',32,[0 0.5 1 1],0,0,0,0,0,0,0);
mglTextDraw('Hello There',[0 0]);
mglFlush;
```

mglStrokeText: Fast no-frills line-based text drawing (does not use texture memory)

purpose: Draws a stroked fixed-width character or string on MGL display. Default width is 1, default height 1.8 (in current screen coordinates)
usage: [x,y]=mglStrokeText(string, x, y, scalex, scaley, linewidth, color, rotation);

argument	value
string	text string. Unsupported characters are printed as #
x,y	center coordinates of first character (current screen coordinates)
scalex	scale factor in x dimension (relative to character) (current screen coordinates). Note that text can be mirrored by setting this factor to a negative value.
scaley	scale factor in y dimension (relative to character) (current screen coordinates). Optional, defaults to scalex.
linewidth	width of line used to draw text. Default 1.
color	text color. Default [1 1 1]
rotation	in radians. Default 0.
OUTPUT x,y	position after last letter (for subsequent calls) [optional]

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglStrokeText('Hello',0,0);
mglFlush;
```

Drawing functions

mglClearScreen

purpose: sets the background color
usage: mglClearScreen(<color>)

argument	value
color	color to set background to, can be a grayscale value or an [r g b] value

set to the level of gray (0-1)

```
mglClearScreen(gray)
```

set to the given [r g b]

```
mglClearScreen([r g b])
```

full example

```
mglOpen;
mglClearScreen([0.7 0.2 0.5]);
mglFlush();
```

mglPoints2: 2D points

purpose: plot 2D points on an OpenGL screen opened with mglOpen
usage: mglPoints2(x,y,size,color)

argument	value
x,y	position of dots on screen
size	size of dots (in pixels)
color	color of dots

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglPoints2(16*rand(500,1)-8,12*rand(500,1)-6,2,1);
mglFlush
```

mglPoints3: 3D points

purpose: plot 2D points on an OpenGL screen opened with mglOpen
usage: mglPoints2(x,y,z,size,color)

argument	value
x,y,z	position of dots on screen
size	size of dots (in pixels)
color	color of dots

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglPoints3(16*rand(500,1)-8,12*rand(500,1)-6,zeros(500,1),2,1);
mglFlush
```

mglLines2: 2D lines

purpose: mex function to plot lines on an OpenGL screen opened with glopen
usage: mglLines(x0, y0, x1, y1,size,color,<bgcolor>)

argument	value
x0,y0	initial position of line
x0,y0	end position of line
size	size of line (in pixels)
color	color of line
bgcolor	background color of line

```
mglOpen
mglVisualAngleCoordinates(57,[16 12]);
mglLines2(-4, -4, 4, 4, 2, [1 0.6 1]);
mglFlush
```

mglFillOval: Ovals

purpose: draw filled oval(s) centered at x,y with size [xsize ysize] and color [rgb]. the function is vectorized, so if you provide many x/y coordinates (identical) ovals will be plotted at all those locations.
usage: mglFillOval(x,y, size, color)

argument	value
x,y	center position of oval
size	[width height] of oval
color	color of oval

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
x = [-1 -4 -3 0 3 4 1];
y = [-1 -4 -3 0 3 4 1];
sz = [1 1];
mglFillOval(x, y, sz,  [1 0 0]);
mglFlush();
```

mglFillRect: Rectangles

purpose: draw filled rectangles(s) centered at x,y with size [xsize ysize] and color [rgb]. the function is

vectorized, so if you provide many x/y coordinates (identical) ovals will be plotted at all those locations.
usage: [ ] = mglFillRect(x,y, size, color)

argument	value
x,y	center position of rectangle
size	[width height] of rectangle
color	color of rectangle

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
x = [-1 -4 -3 0 3 4 1];
y = [-1 -4 -3 0 3 4 1];
sz = [1 1];
mglFillRect(x, y, sz,  [1 1 0]);
mglFlush();
```

mglFixationCross: Cross

purpose: draws a fixation cross with no arguments, draws a fixation cross at origin (default width 0.2 with linewidth 1 in white at [0,0])
usage: mglFixationCross([width], [linewidth], [color], [origin]);
alternate usage: mglFixationCross(params)

argument	value
params	[width linewidth r g b x y]
width	width in degrees of fixation cross
linewidth	width in pixels on line
color	color of fixation cross
origin	center position of fixation (defaults to [0 0])

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
mglFixationCross;
mglFlush;
```

mglGluAnnulus: Annuli, rings

purpose: for annuli and rings, e.g. for retinotopic stimuli. The function is vectorized, such that multiple annuli can be rendered in one call. In this case, x,y, isize, and osize need to have the same number of elements. Color is also vectorized.
usage: [ ] = mglGluAnnulus(x, y, isize, osize, color, [nslices], [nloops])

argument	value
x,y	position of circle from which annulus/annuli is/are derived
isize	inner radius/radii
osize	outer radius/radii

color	color of annuli, either [], 3-vector, or 3-row by n-column matrix
nslices	number of wedges used in polygon→circle approximation [default 8]
nloops	number of annuli used in polygon→circle approximation [default 1]

```
mglOpen(0);
mglVisualAngleCoordinates(57,[16 12]);
x = zeros(10, 1);
y = zeros(10, 1);
isize = ones(10,1)
osize = 3*ones(10,1);
startAngles = linspace(0,180, 10)
sweepAngles = ones(1,10).*10;
colors = jet(10)'; % nb! transpose
mglGluPartialDisk(x, y, isize, osize, startAngles, sweepAngles, colors, 60, 2);
mglFlush();
```

mglGluDisk: Circular dots

purpose: for plotting circular (rather than square dots), use this function. on slower machines, large number of dots may lead to dropped frames. there may be a way to speed this up a bit in future.
usage: [ ] = mglGluDisk(x, y, size, color, [nslices], [nloops])

argument	**value**
x,y	position of dots
size	size of dots
color	color of dots
nslices	number of wedges used in polygon→circle approximation [default 8]
nloops	number of annuli used in polygon→circle approximation [default 1]

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
x = 16*rand(100,1)-8;
y = 12*rand(100,1)-6;
mglGluDisk(x, y, 0.1,  [0.1 0.6 1], 24, 2);
mglFlush();
```

mglGluPartialDisk: Segments, wedges

purpose: for segments and wedges, e.g. for retinotopic stimuli. The function is vectorized, such that multiple segments can be rendered in one call. In this case, x,y, isize, osize, startAngles, and sweepAngles need to have the same number of elements. Color is also vectorized (see mglGluAnnulus and the example below).
usage: [ ] = mglGluPartialDisk(x, y, isize, osize, startAngles, sweepAngles, color, [nslices], [nloops])

argument	**value**
x,y	center position of circle from which segment is derived
isize	innter radius of segment
osize	outer radius of segment

startAngles	angle at which segment(s) start
sweepAngles	angle each segment(s) sweep(s) out
color	color of segment
nslices	number of wedges used in polygon→circle approximation [default 8]
nloops	number of annuli used in polygon→circle approximation [default 2]

```
mglOpen(0);
mglVisualAngleCoordinates(57,[16 12]);
x = zeros(10, 1);
y = zeros(10, 1);
isize = linspace(1, 5, 10);
osize = 3+isize;
startAngles = linspace(0,180, 10)
sweepAngles = ones(1,10).*10;
colors = jet(10)';
mglGluPartialDisk(x, y, isize, osize, startAngles, sweepAngles, colors, 60, 2);
mglFlush();
```

mglPolygon: Polygons

purpose: mex function to draw a polygon in an OpenGL screen opened with mglOpen. x and y can be vectors (the polygon will be closed)
usage: mglPolygon(x, y, [color])

argument	**value**
x,y	position of vertices
color	color of polygon

```
mglOpen;
mglVisualAngleCoordinates(57,[16 12]);
x = [-5 -6 -3  4 5];
y = [ 5  1 -4 -2 3];
mglPolygon(x, y, [1 0 0]);
mglFlush();
```

mglQuads: Quads

usage: mglQuad(vX, vY, rgbColor, [antiAliasFlag]);
purpose: mex function to draw a quad in an OpenGL screen opened with mglOpen

argument	**value**
vX	4 row by N column matrix of 'X' coordinates
vY	4 row by N column matrix of 'Y' coordinates
rgbColors	3 row by N column of r-g-b specifing the color of each quad
antiAliasFlag	turns on antialiasing to smooth the edges

```
mglOpen;
```

```
mglScreenCoordinates
mglQuad([100; 600; 600; 100], [100; 200; 600; 100], [1; 1; 1], 1);
mglFlush();
```

Gamma tables

mglSetGammaTable: Sets the display card gamma table

purpose: Set the gamma table
usage: There are a number of ways of calling this function explained below.

Setting a redMin, redMax, redGamma, greenMin, etc.

```
mglSetGammaTable(0,1,0.8,0,1,0.9,0,1,0.75);
```

or with a vector of length 9:

```
mglSetGammaTable([0 1 0.8 0 1 0.9 0 1 0.75]);
```

or set with a single table for all there colors. Note that the table values go from 0 to 1 (i.e. 0 is the darkest value and 1 is the brightest value). If you have a 10 bit gamma table (most cards do–see section on monitor calibration for a list), then the intermediate values will be interpreted with 10 bits of resolution.

```
gammaTable = ((0:1/255:1).^0.8)';
mglSetGammaTable(gammaTable);
```

or set all three colors with differnet tables

```
redGammaTable = (0:1/255:1).^0.8;
greenGammaTable = (0:1/255:1).^0.9;
blueGammaTable = (0:1/255:1).^0.75;
mglSetGammaTable(redGammaTable,greenGammaTable,blueGammaTable);
```

can also be called with an nx3 table

```
gammaTable(:,1) = (0:1/255:1).^0.8;
gammaTable(:,2) = (0:1/255:1).^0.9;
gammaTable(:,3) = (0:1/255:1).^0.75;
mglSetGammaTable(gammaTable);
```

can also be called with the structure returned by mglGetGammaTable

```
mglSetGammaTable(mglGetGammaTable);
```

Note that the gamma table will be restored to the original after mglClose.

Timing. The setting of the gamma table is done by the OS in a way that seems to be asynchronous with

mglFlush. For instance, the following code gives unexpected results:

```
mglOpen(1);
mglClearScreen(1); % set back buffer to white
mglWaitSecs(2);

% now set the gamma table to all black, this should insure that nothing will be displayed
mglSetGammaTable(zeros(1,256));
mglFlush;
% now the flush will bring the value 255, set by the mglClearScreen above,
% to the front buffer, but because the gamma table is set to black,
% nothing should be displayed

mglWaitSecs(2);
mglClose;
```

This should keep the screen black, but on my machine, the screen temporarily flashes white. Presumably this is because the mglSetGammaTable happens after the mglFlush. It is recommended that you change the gamma while there is nothing displayed on the screen and wait for at least one screen refresh before assuming that the gamma table has actually changed.

mglGetGammaTable: Gets the current gamma table

purpose: returns what the gamma table is set to **usage:** table = mglGetGammaTable()

```
mglOpen;
gammaTable = mglGetGammaTable
```

Stencils to control drawing only to specific parts of screen

Here is a demonstration of how to use stencils using these these functions:

```
mglOpen;
mglScreenCoordinates;
```

```
%Draw an oval stencil
mglStencilCreateBegin(1);
mglFillOval(300,400,[100 100]);
mglStencilCreateEnd;
mglClearScreen;
```

```
% now draw some dots, masked by the oval stencil
mglStencilSelect(1);
mglPoints2(rand(1,5000)*500,rand(1,5000)*500);
mglFlush;
mglStencilSelect(0);
```

mglStencilCreateBegin: Start drawing a stencil

purpose: Begin drawing to stencil. Until mglStencilCreateEnd is called, all drawing operations will also draw to the stencil. Check MGL.stencilBits to see how many stencil planes there are. If invert is set to one, then the inverse stencil is made
usage: mglStencilCreateBegin(stencilNumber,invert)

argument	value
stencilNumber	stencil number, usualy 1-8 but look at the global variable MGL.stencilBits to see how many stencil planes there are.
invert	1 or 0 to invert the stencil that is made

see example above.

mglStencilCreateEnd: End drawing a stencil

purpose: Ends drawing to stencil **usage:** mglStencilCreateEnd

see example above.

mglStencilSelect: Select a stencil

purpose: Sets which stencil to use, 0 for no stencil **usage:** mglStencilSelect(stencilNumber)

argument	value
stencilNumber	number of stencil to use

See example above.

Keyboard and mouse functions

mglDisplayCursor: Hide or display the mouse cursor

purpose: Hide or display the mouse cursor
usage: mglDisplayCursor(<display>)

argument	value
display	1 or 0 to display or hide the cursor

When you call mglOpen the mouse cursor is hidden by default. You can get it to come back by doing:

```
mglOpen
mglDisplayCursor
```

mglGetKeys: Get keyboard state

purpose: returns the status of the keyboard (regardless of whether the focus is on the mgl window)
usage: mglGetKeys(<keys>)

argument	value

keys	array of keycodes. In this case mglGetKeys will only return thestatus of those keys, for example: mglGetKeys([61 46]) will return the values of key 61 and 46.

mglGetMouse: Get mouse state

usage: mglGetMouse()
purpose: returns the status of the mouse buttons (regardless of whether the focuse is on the mgl window

mglGetKeyEvent: Get a key down event off of queue

purpose: returns a key down event waitTicks specifies how long to wait for a key press event in seconds. Note that the timing precision is system-dependent:

- Mac OS X: about 1/60 s
- Linux: 1/HZ s where HZ is the system kernel tick frequency (HZ=100 on older systems, HZ=250 or 500 on more modern systems)

The default wait time is 0, which will return immediately and if no keypress event is found, will return an empty array []. The return structure contains the character (ASCII) code of the pressed key, the system-specific keycode, a keyboard identifier (on Linux, this is the keyboard state, or modifier field), and and the time (in secs) of the key press event. **NOTE** that to get a key event the focus *MUST* be on the mgl window. For faster timing, try mglGetKeys
usage: mglGetKeyEvent(waitTicks)

argument	value
waitTicks	Ticks to wait for before giving up and returning empty event

```
mglOpen
mglGetKeyEvent(0.5)
```

mglGetMouseEvent: Get a mouse button down event off of queue

usage: mglGetMouseEvent(waitTicks)
purpose: returns a mouse down event waitTicks specifies how long to wait for a mouse event in seconds. Note that the timing precision is system-dependent:

- Mac OS X: about 1/60 s
- Linux: 1/HZ s where HZ is the system kernel tick frequency (HZ=100 on older systems, HZ=250 or 500 on more modern systems)

The default wait time is 0, which will return immediately with the mouse position regardless of button state. The return structure contains the x,y coordinates of the mouse, the button identifier if pressed (on the button-challenged Mac this is always 1) and 0 otherwise, and the time (in secs) of the mouse event. **NOTE** that the mouse down event has to be *ON* the mgl window for this to work with waitTicks not equal to 0

argument	value
waitTicks	Ticks to wait for before giving up and returning empty event

```
mglOpen
mglGetMouseEvent(0.5)
```

mglCharToKeycode: Returns keycode of char

Purpose: Returns the keycodes of a (list of) keynames
Usage: keycode=mglCharToKeycode(keyname)

Note on special keys: On Linux (X), special keys and function keys have unique names, e.g., 'Escape', 'F1', etc., so obtaining the keycodes for these is done by mglCharToKeycode({'Escape','F1'}) etc. On Macs, this is not possible; instead, test for the keycode and name of a key using the mglShowKey function.

The keycodes match those used by mglGetKeys and mglGetKeyEvent

argument	value
keyname	cell array where each entry is a key name string e.g. keyname = {'h','g' '1'}
OUTPUT:keycode	vector of integer keycodes for each keyname entry e.g. for the above example, keyname=[44 43 11] (on Linux)

Example: testing for specific keypresses:

```
keycodes=mglCharToKeycode({'1','2' '3'}) % keys 1-3 on main keyboard
while (1); k=mglGetKeys(keycodes); if (any(k)),break;end;end
```

Technical note: the returned keycodes are identical to system keycodes+1

mglKeycodeToChar: Returns char of keycode

Purpose: Returns the keynames of a (list of) keycodes
Usage: keyname=mglKeycodeToChar(keycode)

Note on special keys: This repeats the above entry, deleted.

argument	value
keycode	vector of integer keycodes for each keyname entry for example, keyname=[44 43 11]
OUTPUT:keyname	cell array where each entry is a key name string for the above example on linux keyname = {'h','g' '1'}

Example: testing which keys were pressed:

```
while (1); k=mglGetKeys; if (any(k)),break;end;end
keycodes=find(k);
keynames=mglKeycodeToChar(keycodes)
```

Technical note: keycodes are identical to system keycodes+1

Timing functions

mglGetSecs: Get time in seconds

purpose: Get current or elapsed time
usage: mglGetSecs(<t0>)

argument	value
t0	start time from which to compute elapsed time

To get current time

```
t=mglGetSecs
```

Get elapsed time since t0

```
t0 = mglGetSecs;
elapsedTime=mglGetSecs(t0)
```

mglWaitSecs: Wait for a time in seconds

purpose: Wait for some time
usage: mglWaitSecs(waitTime)

argument	value
waitTime	time to wait for in seconds

Wait 3.3 seconds:

```
mglWaitSecs(3.3);
```

Sound functions

mglInstallSound: Install an .aiff file for playing with mglPlaySound

purpose: Install an .aiff file for playing with mglPlaySound
usage: mglInstallSound(soundName)

argument	value
soundName	aiff filename

This will install sounds to be played using mglPlaySound. Note that if you just want to use systems sounds then you do not need to call this function directly, it will be called by mglOpen to install all your system sounds. Once the sound is installed you can play it with mglPlaySound

```
soundNum = mglInstallSound('/System/Library/Sounds/Submarine.aiff');
mglPlaySound(soundNum);
```

With no arguments, mglInstallSound uninstalls all sounds

```
mglInstallSound
```

mglPlaySound: Play a system sound

purpose: Play a sound
usage: mglPlaySound(soundNum)

argument	value
soundNum	number of sound

Plays a system sound. After calling mglOpen, all of the system sounds will be installed and you can play a specific one as follows:

```
mglOpen;
global MGL;
mglPlaySound(find(strcmp(MGL.soundNames,'Submarine')));
```

With no arguments mglPlaySound plays the system alert sound

```
mglPlaySound
```

Note that this function returns right after it starts playing the sound (it does not wait until the sound finishes playing).

Test/Demo programs

Run these test programs without any parameters and they should display on your second monitor. With an optional single argument you can pass the number of the display you want to display on.

- mglTestAlignment: Alignment of textures
- mglTestDots: Draws dots
- mglTestGamma: GUI controlled gamma
- mglTestLUTAnimation: Gamma LUT animation
- mglTestStencil: Demonstrates stencil functions
- mglTestTex: Draws a gabor
- mglTestTexMulti: Draws many small images to screen
- mglTestText: Draws text
- mglTestKeys: Returns keyboard codes

A quick overview

The task structure can be used to help code experiments, it is completely separate from the basic mgl libarary that is used to display to the screen (in that you do not have to use the task code to use the basic mgl functions).

The structure for these experiments involves three main variables:

myscreen: Holds information about the screen parameters like resolution, etc.
task: Holds info about the block/trial/segment structure of the experiment
stimulus: Holds structures associated with the stimulus.

To create and run an experiment, your program will do the following:

1. Initialize the screen.
2. Set up the task structure. The task structure holds information about the parameters you want to randomize over and the timing of your experiment.
3. Initialize the stimulus. Here you will create all the necessary bitmaps or display structures that you will need to display your stimulus.
4. Create callback functions. These functions will run at various times in the experiment like at the beginning of a trial or when the subject responds with a keypress or before each display refresh. They are the main way that you program how your stimulus will display and what to do when you get subject responses etc.
5. Create a display loop. This is the part that actually runs your experiment. Essentially all you have to do is call updateTask which handles all the hard work of running your task.

The basic idea of how to set up your experiment with these structures requires defining some terms. Going from the largest organization down to the smallest:

- Task: Task refers to the overall experiment. The task is the top level structure. It contains all the parameters that you are testing as well as the information about how the trials are to be run. A task might be the parameters for a set of trials in which you show different visual stimuli. Or a set of trials that run a psychophysical staircase. Note that in some cases you might have more than one task running at the same time. For example, if you are running a retinotopy scan, you may want to have the retinotopic stimuli as one task and a staircased fixation task as the second task.
- Phases: Tasks may sometimes have more than one phase. For example you may want to show an adaptation stimulus for 30 seconds at the beginning of your experiment in one phase, and then go on to the next phase of the experiment in which you will have randomized trials.
- Blocks: A block is a set of trials in which each combination of parameters is presented in one trial. The code takes care of properly randomizing your trials so that in each block of trials each stimulus type is presented once. (You can also choose not to randomize).
- Trials: A single trial of an experiment.
- Segments: Segments divide up the time in a trial. For example you may have one segment with a fixation cross, another segment where the stimulus is presented and a final segment where the subject responds. What each segment does, how many you have and how long they last are all up to you and define how a trial works.

A simple example experiment can be found in mgl/task:

```
testExperiment
```

testExperiment

The code for textExperiment is a good starting place for creating a new experiment since it contains all the essential elements for using these functions.

Let's start by briefly going through each one of the steps above in reference to the function testExperiment. Note that when you actually want to program your own task, you can either start by editing testExperiment.m or use the function taskTemplate.m (be sure to copy these to a new name). taskTemplate.m is an even more stripped down version of testExperiment.m that contains only the necessary essentials to start using the code (and everywhere there is a comment that begins with fix: you will need to make changes to customize for your experiment). There are also some more templates that can be used as starting places:

- taskTemplateStaticStaircase: This is a task that implements a simple staircase task where the stimuli are static and don't need to be updated every screen refresh.
- taskTemplateFlashingStaircase: This is a task that implements a simple staircase task where the stimuli are flashing and need to be updated every screen refresh.
- taskTemplateReactionTime: A simple reaction time task that shows you how to get the most accurate reaction time.
- taskTemplateContrast10bit: Shows you how to use the 10-bit capacity for fine contrast steps
- taskTemplateDualMain: This is an example of the main task in a dual task pair, to show how to run dual tasks.
- taskTemplateDualSubsidiary: This is an example of the subsidiary task in a dual task pair, to show how to run dual tasks.

Initialize the screen

This can be done very simply just by calling

```
% initalize the screen
myscreen = initScreen;
```

This call will handle opening up of the screen with appropriate parameters and setting the gamma table.

If you want to add specific parameters for your computer add a line like the following:

```
myscreen.screenParams{1} = {'yoyodyne.cns.nyu.edu',[],2,1280,1024,57,[31 23],60,1,1,1.8,'calibFilename
myscreen = initScreen(myscreen);
```

This will set parameters for your screen. The parameters in order are

- computerName
- displayName (optional-for computers with multiple displays like lcd and projector)
- displayNumber
- screenWidth (in pixels)
- screenHeight (in pixels)
- displayDistances (in cm)
- displaySize (in cm)
- framesPerSecond (in Hz)
- autoCloseScreen (1 to close screen at end of experiment, 0 to leave it open)
- saveData (1 to save data file, 0 not to save data file,n>1 saves a data file only if you exceed n number of volumes)
- monitorGamma (The monitor gamma to correct for if you do not have a calibration file. Macs are supposed to have a gamma of 1.8)
- calibFilename (the name of the calibration file-usually just the computer name-see below under moncalib)
- flipHV (Whether to flip the screen horizontally and/or vertically-an array of length two 0=no flip, 1 = flip)

Setup the task structure

In the testExperiment, the task structure is a cell array that actually contains two separate tasks that will be run in the course of the experiment.

This sets the first task to be the fixation staircase task. If you don't want to use the fixation task then you can omit this part:

```
% set the first task to be the fixation staircase task
[task{1} myscreen] = fixStairInitTask(myscreen);
```

This is the first “phase” of our task. Not all tasks need to have different phases, but in this case we want the experiment to start with dots moving incoherently for 10 seconds and then we want trials to run in the next phase.

```
% set our task to have two phases.
% one starts out with dots moving for incohrently for 10 seconds
task{2}{1}.waitForBacktick = 1;
task{2}{1}.seglen = 10;
task{2}{1}.numBlocks = 1;
task{2}{1}.parameter.dir = 0;
task{2}{1}.parameter.coherence = 0;
```

Each one of the fields in the task set the behavior of that phase of the task.

- waitForBacktick=1: The task phase will only start running after we receive a keyboard backtick (`).
- seglen = 10: The segment will run for 10 seconds.
- numBlocks = 1: There will be one block of trials before we run on to the next phase of the task.
- paramater.dir = 0: We set the parameter dir to have a value of 0.
- parameter.coherence = 0: We set the parameter coherence to have a value of 0.

The next phase of the task will be the one that actually runs the trials.

```
% the second phase has 2 second bursts of directions, followed by
% a top-up period of the same direction
task{2}{2}.segmin = [2 6];
task{2}{2}.segmax = [2 10];
task{2}{2}.parameter.dir = 0:60:360;
task{2}{2}.parameter.coherence = 1;
task{2}{2}.random = 1;
% and set to remember the values for dir
task{2}{2}.writetrace{1}.tracenum = 1;
task{2}{2}.writetrace{1}.tracevar{1} = 'dir';
task{2}{2}.writetrace{1}.usenum = 1;
```

In this task, we have a block of trials in which we will show trials with different motion directions. You set what parameters you want to use in the “parameter” part of your task. Note that you can use any name for parameters that you like. Here we call them dir for direction and coherence for motion coherence. Note that we have only one value of motion coherence so all trials will be run with a motion coherence of 1.

```
task{2}{2}.parameter.dir = 0:60:360;
task{2}{2}.parameter.coherence = 1;
```

We also have to decide the order in which parameters will be presented in a block of trials. The default is to run

them sequentially (in this case directions 0 then 60 then 120 etc). To randomize the order, we set:

```
task{2}{2}.random = 1;
```

Our trial will have two segments, a 2 second segment in which the stimulus is presented and a 6-10 second long intertrial interval:

```
task{2}{2}.segmin = [2 6];
task{2}{2}.segmax = [2 10];
```

Finally, to keep track of what direction was shown on one trial we can keep a “trace” of the dir parameter. This is not the ony way to get the information about what was shown on what trial, but it is fairly convenient. A trace will start with the value 0 and then on the segment of our choosing will change to the value of the parameter that was presented on the trial. By plotting the trace you can see the timing of your trials and the parameter that was chosen. In this case, we want to write out the direction parameter on the first segment of the trial, so we have writetrace{1} (if we wanted the second segment we would do writetrace{2} etc). We are going to use the 1st trace to store our information (you can have as many traces as you want to track different variables). We want to save the parameter 'dir' and instead of writing out 0, 60, 120 etc. we “usenum” which means that we will write out the corresponding number (i.e. 1, 2, 3 etc) used to represent the parameter.

```
% and set to remember the values for dir
task{2}{2}.writetrace{1}.tracenum = 1;
task{2}{2}.writetrace{1}.tracevar{1} = 'dir';
task{2}{2}.writetrace{1}.usenum = 1;
```

Initialize the stimulus.

The stimulus is kept in a global variable so that if the variable is very large, we don't incur overhead with passing it around all the time. If you want to have the stimulus variable saved at the end of the experiment, you can call the function initStimulus as below. Note that you do not need to call initStimulus if you do not want to save the stimulus structure.

```
% init the stimulus
global stimulus;
myscreen = initStimulus('stimulus',myscreen);
stimulus = initDots(stimulus,myscreen);
```

The function initDots is specific for creating the dots stimulus for this test experiment, you will substitute your own function for creating your stimulus.

Create callback functions

Callbacks are the way that you control what happens on different portions of the trial and what gets drawn to the screen. A callback is simply a function that gets called at a specific time. You write the function and you let updateTask handle when that function needs to be called.

There are two required callbacks:

The first required callback that is used in this program is the one that gets called every time a segment starts.

```
function [task myscreen] = startSegmentCallback(task, myscreen)
```

```
global stimulus;
if (task.thistrial.thisseg == 1)
  stimulus.dots.coherence = task.thistrial.coherence;
else
  stimulus.dots.coherence = 0;
end
stimulus.dots.dir = task.thistrial.dir;
```

What it does is it looks in the “thistrial” structure for what segment we are on, if we are not in segment one (i.e. the intertrial interval) it sets the motion coherence to 0, otherwise it sets it to whatever the parameter coherence is set to (defined in the task.parameter.coherence field). It also sets the direction of motion of the dots.

The second (and most important) callback is the one used to draw the stimulus to the screen:

```
function [task myscreen] = screenUpdateCallback(task, myscreen)
```

```
global stimulus
mglClearScreen;
stimulus = updateDots(stimulus,myscreen);
```

You can put your stimulus drawing routines in here. In this program, we simply clear the screen and draw the dots. This function gets called every display refresh.

Once these functions are defined in your file, you tell the programs to use these callbacks by using initTask to register the callbacks.

```
% initialize our task with only the two required callbacks
for phaseNum = 1:length(task{1})
  [task{1}{phaseNum} myscreen] = initTask(task{1}{phaseNum},myscreen,@startSegmentCallback,@screenUpda
end
```

NOTE: It is necessary to register the callbacks in a specific order. The correct order for registering callbacks is: startSegmentCallback, screenUpdateCallback, getResponseCallback, startTrialCallback, endTrialCallback, startBlockCallback

It doesn't matter exactly how you name the callbacks, what matters is what order you call them in. If there is a callback that you are not defining, you can enter it as [] in the initTask call, or leave it out:

for example,

```
[task myscreen] = initTask(task,myscreen,@startSegment, @screenUpdate, @getResponse, [],[], @startBloc
```

or

```
[task myscreen] = initTask(task,myscreen, @startSegment, @screenUpdate, @getResponse);
```

More details can be found in the callbacks section.

Create a display loop

Now that everything is setup to run your experiment all you need is a display loop that calls updateTask to run each one of the tasks that are being displayed. Then to flip the front and back buffer of the display to show your stimulus, you call tickScreen. This is the main loop in which your program is run.

```
phaseNum = 1;
while (phaseNum <= length(task{2})) && ~myscreen.userHitEsc
  % update the dots
  [task{2} myscreen phaseNum] = updateTask(task{2},myscreen,phaseNum);
  % update the fixation task
  [task{1} myscreen] = updateTask(task{1},myscreen,1);
  % flip screen
  myscreen = tickScreen(myscreen,task);
end
```

At the very end you end the task which will save out information about your experiment.

```
myscreen = endTask(myscreen,task);
```

Experimental parameters

Basics

For your experiment you can choose what parameters you have and what values they can take on. You do this by adding parameters (of your choosing) into the parameter part of a task variable:

```
task.parameter.myParameter1 = [1 3 5 10];
task.parameter.myParameter2 = [-1 1];
```

You can add any number of parameters that you want. updateTask will chose a value on each trial and put those values into the thistrial structure:

```
task.thistrial.myParameter1
task.thistrial.myParameter2
```

would equal the setting on that particular trial. In each block every combination of parameters will be presented. You can randomize the order of the parameters by setting:

```
task.random = 1;
```

Note that parameter should really just be used for the parameters over which you want to randomize your experiment. For example, you may be testing several contrasts in your experiment, that should be coded as a parameter. You may also have some random variables, things like which segment that target should be presented in for example–things that need to be randomized, but are not a crucial parameter you are testing. For these types of variables, you should use randVars instead of parameter (see below).

What if I have a group of parameters

You may have stimuli in which the parameters are grouped into different sets. For example you might want to show two types of grating patches. One tilted to the left with a high contrast and low spatial frequency and the other tilted to the right with low contrast and high spatial frequency.

Then you could do

```
task.parameter.groupNum = [1 2];
task.private.group{1}.orientation = -10;
task.private.group{1}.contrast = 1;
task.private.group{1}.sf = 0.2;
task.private.group{2}.orientation = 10;
task.private.group{2}.contrast = 0.1;
task.private.group{2}.sf = 4;
```

On each trial, you get the parameters by doing

```
task.thistrial.thisgroup = task.private.group{task.thistrial.groupNum};
```

What if I have parameters that are not single numbers

You may have a parameter that is an array rather than a single number. Again, do something like the above (1.3)

```
task.parameter.stringNum = [1 2 3];
task.strings = {'string1','string2','string3'}
```

and get the appropriate string on each trial by doing:

```
task.thistrial.thisstring = task.strings{task.thistrial.stringNum};
```

randVars

For variables that you just want to have some randomization over, you can declare them as randVars. For example, you might want to specify a target interval which should be either 1 or 2 on any given trial, but you don't want that to be block randomized. Then you can declare that variable as a uniform randomization:

```
task.randVars.uniform.targetInterval = [1 2];
```

This variable will then be available in task.thistrial.targetInterval.

You may also want to have the variable block randomized, like a parameter, but the blocks should be independent of the main parameter:

```
task.randVars.block.blockedVar = [-1 0 1];
```

This will guarantee that on every three trials, blockedVar will be set to each one of the possible values -1,0 and 1.

Note that with randVars the randomization is chosen at the beginning of the experiment and by default 250 trials are randomized after which you will cycle back through the variables. If you need more than 250 trials, you can set:

```
task.randVars.len_ = 500;
```

Using your own random sequence

You might have your own randomization routine and want to use that to randomize parameters. You can do that with randVars:

```
task.randVars.myRandomParameter = [...];
```

Then myRandomParameter will be available in task.thistrial.myRandomParameter in the order you specify in the array.

Segment times

How to setup segment times

Each trial can be divided into multiple segments where different things happen, like for instance you might have a stimulus segment and response segment that you want to have occur for 1.3 and 2.4 seconds respectively:

```
task.seglen = [1.3 2.4];
```

At the beginning of each segment the callback startSegment will be called and you can find out which segment is being run by looking at:

```
task.thistrial.thisseg
```

How to randomize the length of segments

If you want to randomize the length of segments over a uniform distribution, like for instance when you want the first segment to be exactly 1.3 seconds and the second segments to be randomized over the interval 2-2.5 seconds:

```
task.segmin = [1.3 2];
task.segmax = [1.3 2.5];
```

In this case, do not specify task.seglen.

If you want the second interval to be randomized over the interval 2-2.5 seconds in intervals of 0.1 seconds (i.e. you want it to be either 2,2.1,2.2,2.3,2.4 or 2.5:

```
task.segmin = [1.3 2];
task.segmax = [1.3 2.5];
task.segquant = [0 0.1];
```

You can also have a segment wait until a backtick happens, so that you can easily synch to volumes, for example:

```
task.segmin = [1.3 2];
task.segmax = [1.3 2.5];
task.synchToVol = [0 1];
```

This will cause the second segment to last a random amount of time between 2 and 2.5 seconds and then wait until a backtick occurs before going on to the next trial.

How to wait for user input before moving to next segment

Sometimes you will want to wait for user input to decide when to end a segment of the trial, rather than pre-set a time. To do this, you need to: (1) set the segment length to inf, (2) take user input for that segment, and (3) in the responseCallback, end the segment when the subject responds. [Note that if you want to limit how much time the user has to respond, but still wait for input, you can set the segment length to something less than inf, e.g. 5 seconds; this means that the segment will end either when the subject responds, or when 5 seconds have elapsed, whichever comes first.]

An example of how this might be implemented, in the case when the second of three segments waits for subject input before terminating:

```
% in the main task body:
task.seglen = [.5 inf 2];
task.getresponse = [0 1 0];
```

```
% At the end of the responseCallback function:
task = jumpsegment(task);
```

For other uses of jumpsegment, and for how to use jumpsegment(task, inf), see how to program a dual task below.

Keeping time in seconds, volumes or refreshes

Trial segments can keep time in either seconds (default), volumes or monitor refreshes.

To change timing to use volumes:

```
task.timeInVols = 1;
```

To change timing to use monitor refreshes (note that is probably not a great idea to keep time in monitor

refreshes since if you drop a frame, your timing will be altered).

```
task.timeInTicks = 1;
```

With timeInVols or timeInTicks, your segment times should now be integer values that specify time in Vols or monitor refreshes (e.g.):

```
task.seglen = [3 2];
```

Note, that the default (time in seconds) adjusts for segment overruns that might occur when you drop monitor frames, but the timeInTicks will not and is therefore usually less accurate.

Callbacks

Callbacks are the way that you control what happens on different portions of the trial and what gets drawn to the screen. They are simply functions that get called at specific times in the experiment.

It doesn't matter exactly what you call them, but it does matter exactly what order you register them in.

There are two required callbacks, and the rest are optional. If for some reason you don't need one of the required callbacks, you can just leave it empty, but you must still define it.

Callbacks are also discussed in the overview.

Registering callbacks

You must register your callbacks with the initTask function, in the following order:

```
[task myscreen] = initTask(task,myscreen,@startSegmentCallback,@screenUpdateCallback,@getResponseCallb
```

You do not need to specify all the callbacks, only startSegmentCallback and screenUpdateCallback. To omit any of the callbacks, either don't pass it in to initTask or set the appropriate argument to []. Make sure that you return task and myscreen.

For example, you might have

```
[task myscreen] = initTask(task,myscreen,@startSegmentCallback,@screenUpdateCallback,[],@startTrialCal
```

or

```
[task myscreen] = initTask(task,myscreen,@startSegmentCallback,@screenUpdateCallback,@getResponseCallb
```

screenUpdateCallback (required)

```
function [task myscreen] = screenUpdateCallback(task, myscreen)
% do your draw functions in here.
```

Note that you will normally declare a global variable named stimulus that contains any textures or information about the stimulus and use that in here. Remember that screenUpdateCallback gets called **every** frame update. For a refresh rate of 60 Hz that means it definitely has to run within 1/60 th of a second, or else the program will start to drop frames and become slow. You should therefore make this function as simple as possible. For example, if you are using textures, call mglCreateTexture in your myInitStimulus function and only use the precomputed texture here in an mglBltTexture function.

Another option that you can consider is that for many types of stimulus you don't have to update the screen every frame refresh. For something like moving dots or a drifting gabor you will need to update the frame every screen refresh, but if you just want to show a static gabor for a full segment, you can use the flushMode=1 feature that is described below in startSegmentCallback.

startSegmentCallback (required)

The other mandatory callback is the one that is called at the beginning of each segment:

```
function [task myscreen] = startSegmentCallback(task, myscreen)
```

The variable task.thistrial will have fields set to what the parameters are for that trial. For instance if you have dir as one of your parameters, then you will have the field task.thistrial.dir set to one of the directions (chosen by updateTask).

If you are only drawing to the screen at the start of every segment, then you can use the flushMode=1 feature. Say for example you want to clear the screen and draw your texture to the screen and that is all that will happen in the segment then you can do something like:

```
mglClearScreen;
mglBltTexture(stimulus.tex,[4 0]);
myscreen.flushMode = 1;
```

Note that in this case you do **not** do any drawing in the screenUpdateCallback (this function will be empty). You only do drawing in the startSegmentCallback. This assumes that the only time the screen changes is when you start a new segment of your trial.

getResponseCallback (optional)

You can (optionally) define a callback for when the subject hits a response key:

```
function [task myscreen] = getResponseCallback(task,myscreen)
```

If you don't have subject responses in your experiment, you can just put this one line in with nothing after it.

There is a field called

```
task.thistrial.whichButton
```

This will get filled with which button was pressed (a number from 1-9). Note that if two keys are pressed down at the same time, it will only return the first in the list (e.g. if 1 and 2 are simultaneously pressed, it will return

1).

If you want to get all the keys that are pressed, you can look at

```
task.thistrial.buttonState
```

This will be an array where each element will have 0 or 1 depending on whether the key was pressed or not.

Note that the getResponseCallback will **only** be called if in the task structure you have set the appropriate segment of the getResponse variable. For example, if you have a two segment trial, and you want to get subject responses in the second segment of the trial you would do:

```
task.getResponse = [0 1];
```

You may also set a getResponse segment to 2. What this does is similar to setting myscreen.flushMode = 1. It prevents mglFlush from being called to update the screen while you are waiting for a keyboard press. This will get much more accurate keyboard timing, but will not allow the screen to update while you are waiting (i.e. you have to have a static display–no moving dots or flickering gratings or anything).

```
task.getResponse = [0 2];
```

If you want to get other keys, rather than the defined keys 1–9, for example if you want the keypad numbers, you can override which keys will be checked with:

```
myscreen.keyboard.nums = [84 85];
myscreen = initScreen(myscreen);
```

This is called at the beginning of your program. Note that to get the keycodes that correspond to a key, you can either use:

```
mglCharToKeycode({'a' 'b' 'c'})
```

or, for keys that you can't write like the keypad numbers or the esc key, run the program:

```
mglTestKeys
```

and type the keys you want and it will print out the correct keycode.

The getResponseCallback will get called every time the subject presses a button, so if the subject presses two buttons one after the other during the response period, getResponseCallback will be called twice. If you want to ignore the 2nd button press you can do:

```
if task.thistrial.gotResponse == 0
  %your response code here
end
```

task.thistrial.gotResponse will be set to 1 the second time the subject presses a key.

startTrialCallback (optional)

You can (optionally) define a callback that gets called at the beginning of each trial

```
function [task myscreen] = startTrialCallback(task,myscreen);
```

endTrialCallback (optional)

You can (optionally) define a callback that gets called at the end of each trial

```
function [task myscreen] = endTrialCallback(task,myscreen);
```

startBlockCallback (optional)

You can (optionally) define a callback that gets called at the beginning of a block

```
[task myscreen] = startBlockCallback(task,myscreen)
```

Saving data

stim files

After you have run an experiment, all three variables (myscreen, task and your stimulus variable) will get saved into a file called

```
yymmdd_stimnn.mat
```

Where yymmdd is the current date, and nn is a sequential number starting at 01. This file will be stored in the current directory or in the directory ~/data if you have one.

After these get saved, you can access all the variables for your experiment by using

```
getTaskParameters(myscreen,task);
```

This will return a strucutre that contains the starting volume of each trial, what each variable was set to, the response of the subject and reaction time, among other things. For most purposes this should contain all the information you need to reconstruct what was presented on what trial and what the subject's response was.

Note that there is a variable called myscreen.saveData which tells the task structure whether to save the stim file or not. The default on your computer is probably set **not** to save the stim file. When you run on the computer in the scanner room, it will save the file automatically. For debugging purposes this is usually what you want so that you don't save unnecessary stim files every time you test your program. However if you want to save the stim file on your test computer to look at, you can add the following to your code where you call initScreen:

```
myscreen.saveData = 1;
```

```
myscreen = initScreen(myscreen);
```

The variables stored in the stim file contain all the information you should need to recreate what happened in your experiment. In fact, it even contains a full listing of the file you used when running the experiment. This is useful since often you might make minor changes to the program and forget what version you were using when you ran an experiment. You can access a listing from the task variable:

```
task{1}{1}.taskFileListing
```

You can also access different aspects of your task variables with the following helper functions:

getTaskParameters

Gets all the info about your task and its parameters:

```
e = getTaskParameters(myscreen,task);
```

getTaskVarnames

Gets a cell array of the variables names in your task

```
varnames = getTaskVarnames(task);
```

getParameterTrace

Gets a trace of the variable called for

```
plot(getParameterTrace(myscreen,task,'varname');
```

getStimvolFromVarname

Gets a cell array that contains the stimulus volumes for a particuar variable name

```
stimvol = getStimvolFromVarname(varname,myscreen,task);
```

getVarFromParameters

Gets the variable settings for each trial

```
getVarFromParameters('varname',getTaskParameters(myscreen,task));
```

Traces

For most people, using getTaskParameters is the easiest way to get what happened on each trial. But there is

another mechanism that allows you to see the specific timing of events as traces. This is saved in the traces field of the myscreen variable. This field stores when each volume was collected and what stimulus was presented. Using this information you can reconstruct the volume when each stimulus occurred. It is set up so the first row contains an array which has a one every time a volume was acquired (i.e. whenever a backtick was received) and zeros elsewhere. The timebase for the array is in monitor refreshes, so every 60 elements shouls be one second. Take a look at what this trace has by doing:

```
myscreen = makeTraces(myscreen);
plot(myscreen.traces(1,:));
```

You can also plot in seconds, relative to the beginning of the experiment:

```
plot(myscreen.time,myscreen.traces(1,:));
```

The other important trace is the one corresponding to myscreen.stimtrace:

```
plot(myscreen.traces(myscreen.stimtrace,:));
```

This will contain the information about which trial was presented as long as you have set the writeTrace variable correctly (see next section).

writeTrace

The following information is only useful for people who need to save extra information in the traces.

Saving your own variables

If you want to save information on values that you calculate yourself, you will call the function writeTrace. The syntax is:

```
myscreen = writeTrace(data,tracenum,myscreen,force);
```

where data is the scalar value you want to save. Tracenum is the trace you want to save to. Note that the first tracenum from the above section is actually saved to myscreen.stimtrace which is usually set to 5. Therefore you will want to save in some trace above myscreen.stimtrace–for example myscreen.stimtrace+1. You will usually want to set force = 1, see the help on writeTrace if you need more information.

This writeTrace function can be called anywhere in your code, for example in the startSegmentCallback. If you had a variable called myParam set to some value you want to save, you will add the code:

```
myscreen = writeTrace(myParam,myscreen.stimtrace+1,myscreen,1);
```

Then after calling getTaskParameters, your variable settings will be available in the traces field.

choosing a directory

By default, mgl will save the data in ~/data if that directory exists, and in the current directory if ~/data doesn't

exist. To save data to a specific directory instead of to these defaults, set

```
myscreen.datadir = datadirname;
```

where datadirname is the full path of the desired directory.

How-Tos

How to end the experiment

In general, the easiest way to code the stimulus is to have it continue indefinitely until the scanner stops scanning. After the scan is finished and you want to stop the stimulus you hit the ESC key. This way you never have the stimulus stop before the scanner does, and it doesn't hurt to keep having the stimulus go past the end of the scan.

If instead you want to only collect a specific number of blocks of trials and stop, then you would set:

```
task{1}.numBlocks = 4;
```

say, to run for 4 blocks of trials and then stop. Or if you want to run for a specific number of trials and stop, then you can do:

```
task{1}.numTrials = 17;
```

which would run for 17 trials and stop. These variables default to inf so that the experiment only stops when the user hits ESC.

How to use 10-bit contrast

If you want to use 10-bits so as to be able to display finer contrast gradations, you need to remap the usual 8-bit contrast steps (0:255) into a subset of the larger 10-bit (1024) contrast table. This can be done using a piece of code called setGammaTable that can be included in your code as a subfunction (written by JG and FP and found at ~shani/matlab/MGLexpts/setGammaTable.m), but there are some details to be careful of.

First, you will want to 'reserve' some colors that you will want to be able to use and leave unaffected by the resetting of the gamma table. This allows you to show, for example, a high-contrast fixation cross at the same time that you're showing a low-contrast target. If you don't reserve some colors, you won't be able to have anything high-contrast at the same time as you use the 10-bit capacity. See example code taskTemplateContrast10bit.m where four colors are saved, and a low-contrast target is shown (written by SO and found at mgl/task/taskTemplateContrast10bit.m).

How to run a dual task

If you want to run two tasks at once, for example, an RSVP task at fixation and a detection task in the periphery, you will create two tasks and call one from within the other. You should construct it so one task (e.g. detection)

is the main task and the other task (e.g. fixation-RSVP) is the subsidiary task.

The subsidiary task needs to be constructed like a regular task, with its own initialization and callbacks, but without the updateTask loop. It will be updated from within the main task.

The main task will be constructed as usual, but an extra line will appear to set the subsidiary task and to update it. For example, to set the fixation task as the subsidiary, you will add a line in the main task like this:

```
task{2} = fixationTask(myscreen);
```

Then, the update loop of the main task will look like this:

```
phaseNum = 1;
while (phaseNum <= length(task{1})) && ~myscreen.userHitEsc
 % update the task
 [task{1} myscreen phaseNum] = updateTask(task{1},myscreen,phaseNum);
 [task{2} myscreen] = updateTask(task{2},myscreen,1);
 % flip screen
 myscreen = tickScreen(myscreen,task);
end
% if we got here, we are at the end of the experiment
myscreen = endTask(myscreen,task);
```

The key to getting this to work is to control the timing. One way to do this is to have the main task set some variables which tell the subsidiary task whether or not to run. In order to do this, have the stimulus variable set as a global variable in both tasks. Set two stimulus subfields as flags, e.g. stimulus.startSubsidiary and stimulus.endSubsidiary, in order to control the subsidiary task. Then have the subsidiary task check the status of these flags, and start or stop accordingly.

In order to get the subsidiary task to start and stop when the appropriate flags are set, you will need to do the following:

Set the first segment of the subsidiary task to have infinite length. That makes the subsidiary wait in the first segment until the main task calls it. When the main task wants to start the subsidiary task, it will set the stimulus.startSubsidary flag to 1, and this will cause the subsidiary to jump to the next segment as follows:

In the screenUpdate callback of the subsidiary task, have a loop that checks to see whether the stimulus.startSubsidiary flag is set to 1. (This should be done in screenUpdate so that it can check all the time.) Have an if-loop that tells the task to skip ahead to the next segment as soon as the flag == 1. (It's a good idea to reset the flag to 0):

```
if(stimulus.startSubsidiary == 1)
 stimulus.startSubsidiary = 0;
 task = jumpSegment(task);
end
```

When you're ready to end the subsidiary task, have the main task set the stimulus.endSubsidiary flag to 1, and have the following if-loop in the subsidiary's screenUpdate callback:

```
if(stimulus.endSubsidiary == 1)
 stimulus.endSubsidiary = 0;
 task = jumpSegment(task,inf);
```

```
end
```

The ‘inf’ argument in the jumpSegment function call tells the task to jump to the end of all the segments and start the next trial. This puts the subsidiary task back into the state of being in the infinite first segment, waiting for the start flag to be reset to 1 by the main task.

Example code can be found in taskTemplateDualMain.m and taskTemplateDualSubsidiary.m

How to calibrate the monitor

Moncalib

To calibrate a monitor, you can use the program moncalib.m in the utils directory. It is set up to work with the PhotoResearch PR650 photometer/colorimeter (which the Lennie lab has) and a serial port adaptor (use the one from the Carrasco lab it is a white Keyspan USA-28 and says Carrasco Lab on it–the one that is in the bag with the photometer is a white translucent Keyspan USA-28X B and doesn't seem to work properly). The serial port interface for matlab is included in the mgl distribution but can also be found on the Mathworks website [1 [http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=4952&objectType=file]]. To use the Keyspan USA-28 adaptor you will need to download a driver from [2 [http://www.keyspan.com/downloads/homepage_pn_usa28.spml]].

- Tricky–When using the automated calibration via the serial port, the program will ask you to turn on the PR650 and then press 'return' within 5 secs. You might not want to press 'return' right away, or you may get something like this on the photometer:

```
PR650 REMOTE MODE
(XFER) s/w ver 1.02
CMD 51 NAK
```

This indicates that you pressed the return while the photometer is waiting for a transfer signal (not sure what it is), and hence entered the XFER mode. If you wait another 2 secs or so it will enter the control mode, now press 'return' you should see this:

```
PR650 REMOTE MODE
(CTRL) s/w ver 1.19
CMD B
```

Basically there is about 2-3 secs time window you should press 'return' to get to this state.

- Tricky2–When doing the automated calibration, turn off screensavers and energysaver, otherwise the screen will go blank after a while and you'll be measuring luminance of blank sreens.

If you cannot install the serial port interface or don't want to automatically calibrate using the USB cable you can also use the program to run manually with any photometer by typing in the luminance measurements yourself.

The program moncalib will save a calibration file in the local directory. For you to use this calibration file, you can store it in one of two places. Either in your own program directory under a directory called displays:

```
./displays
```

Or you can store it in the general displays directory

```
mgl/task/displays
```

InitScreen should automatically find the correct table by checking your computer name and looking for the file in these two places. If you do not use the standard filename, or have multiple calibrations for the same computer (like if you have multiple monitors calibrated), you can use a specific file by setting myscreen.calibFilename

```
myscreen.calibFilename = 'mycalibrationfile.mat';
myscreen = initScreen(myscreen);
```

Note that the calibFilename can be a literal filename as in the above, or you can specify a portion of the name that will get matched in a file from the displays directory (e.g. computername_displayname would matcha any file in the displays directory that looks like *computername_displayname*.mat).

The name of the file usually created by moncalib will be:

```
xxxx_computername_yymmdd.mat
```

Where xxxx is a sequential number starting at 0001 and yymmdd is the date of the calibration. This stores a variable called calib which contains all the information about the calibration. You can quickly plot the data in calib by doing:

```
load 0001_stimulus-g5_LCD_061004
moncalib(calib);
```

The most important field of calib is the table field which holds the inverse lookup table to linearize the monitor.

10 bit gamma tables

The NVIDIA GeForce series of video cards have 10 bit gamma tables (these are the only ones we have tested):

- NVIDIA GeForce FX 6600 (In the G5 in the magnet room)
- NVIDIA GeForce FX 7300 GT (brownie Mike Landy's psychophysics room)
- NVIDIA GeForce FX ????? (Jackson the G5 in the psychophysics room)

ATI 10 bit cards:

- any Randeon card for desktop computers above series 7000 has 10-bits DAC resolution (laptop cards don't have it necessarely or drivers do not access it)
- some more information about this can be found on Denis Pelli webpage [3 [http://vision.nyu.edu/Tips/VideoCards.html#morebits]] and on the Psychtoolbox.org discussion group [4 [http://psychtoolbox.org/PTB-2/mac.html#drivers]].

It is always the best to use the bit test in moncalib because some drivers do not allow 10-bit control on 10-bit DAC cards. You can also query the display card to see if it says that it supports a 10 bit gamma:

```
displayInfo = mglDescribeDisplays
```

Check the field gammaTableWidth to see if it is 10.

Calibration devices

Note that there are some commercially available devices to calibrate monitor screens which create color profiling information (e.g. [5 [http://usa.gretagmacbethstore.com/index.cfm/act/Catalog.cfm/catalogid/1861/Subcategory/Eye-One%20Solutions/category/Eye-One/browse/null/MenuGroup/_Menu%20USA%20New/desc/Eye-One%20Display%202.htm]] [6 [http://www.xritephoto.com/product/optixxr/]] [7 [http://www.colorvision.com/]]. We have tested one of these called Spyder2Pro which allows you to linearize the monitor output but found that is not yet suitable for psychophysics purposes. The calibration program crashes when you use the default settings to linearize the monitor (an email to the tech support confirmed this is a bug in their software). Using advanced settings it worked but it could only test luminance at 5 output levels. The linearization that it achieved was not accurate enough when tested with the PR650 (it looked like they are doing some sort of spline fit of the points and the luminance as a function of monitor output level looked like a wavy line around the ideal).

How to run an experiment with the same random sequence as a previous one

You can do this by calling initScreen with the randstate of the previous experiment

```
initScreen([],previousMyscreen.randstate);
```

This will insure that all the parameters, randVars and segment times are generated with the same random sequence as the previous experiment.

Alternatively, you can run both experiments starting with the same randstate (which can be an integer value). For example

```
initScreen([],11);
```

Will run the experiment with exactly the same randomization sequence every time.

Task function reference

initScreen

purpose: initializes the screen
usage: myscreen = initScreen(<myscreen>,<randstate>)

argument	value
myscreen	Contains any desired initial parameters, can be left off if you are just using all defaults
randstate	Sets the initial status of the random number generator. This can either be an integer value, or it can be the field myscreen.randstate to set the state back to what it was on a particular experiment.

This function initializes the screen by calling mglOpen, and also handles a number of different default initialization procedures such as setting up the gamma table with the correct linearization table. You should call this **once** at the beginning of the experiment. The variable myscreen will contain many fields associated with the status of the screen and records events like volume acquisitions and trial/segment times etc.

initStimulus

purpose: initializes the global stimulus variable name
usage: myscreen = initScreen('stimulusName',myscreen);

argument	value
stimulusName	string that contains the name of the global variable that is used for your stimulus (i.e. if you had global stimulus, then this should be 'stimulus')
myscreen	myscreen variable returned by initScreen

Note that this function, **only** needs to be called if you want to save the stimulus in your stim file. Since stimulus is a global variable, if you call this function, at the end of the experiment it will get the global variable with the name you specified here and save it in your stim file. If you do not need to save your stimulus variable, you do not need to call this function.

initTask

purpose: initializes a task variable
usage: [task myscreen] = initTask(task,myscreen,startSegmentCallback,screenUpdateCallback, trialResponseCallback, <startTrialCallback>, <endTrialCallback>, <startBlockCallback>)

argument	value
task	Parameters for the particular task (note this must be a struct **not** a cell array, for a cell array, call initTask fore each element of the cell array.
myscreen	Variable returned by initScreen
startSegmentCallback	Function pointer that will be called at start of a segment
screenUpdateCallback	Function pointer that will be called every screen update (i.e. for a 60Hz buffer once every 1/60 of a second)
trialResponseCallback	Function pointer that will be called when the subject responds and getResponse is set
startTrialCallback	Function pointer that will be called at start of a trial
endTrialCallback	Function pointer that will be called at end of a trial
startBlockCallback	Function pointer that will be called at start of a block

The task variable gets set up as explained above. Here is a list of valid fields:

field	value
verbose	display verbose message when running tasks (probably shouldn't be set for real experiment since print statements can be slow)
parameter	task parameters
seglen	array of length of segments (used when not using segmin and segmax)
segmin	array of minimum length of segment
segmax	array of maximum length of segment
segquant	array of quantization of segment lengths (used with segmin and segmax–i.e. if you want segments to be randomized in steps of 0.6 seconds, then set the sequant for that segment to be 0.6)
synchToVol	array where one means that the segment will synch to the next volume acquisiton once the segment is finished.
writeTrace	traces to write out (usually internal variable, that you do not have to set)
getResponse	array where one means to get subject responses during that segment, set to zero means that subject responses will be ignored and the responseCallback will not be called
numBlocks	number of blocks of trials to run before stopping
numTrials	number of trials to run before stopping

waitForBacktick	wait for a backtick before starting task phase
random	randomize the order of parameters for each trial when set to 1, otherwise have the parameters go in order
timeInTicks	when set to 1, segment legnths are in screen updates (not in seconds)
timeInVols	when set to 1, segment lengths are in volumes (not in seconds)
segmentTrace	internal variable that controls what trace this task will use to save out segment times (usually you will not set this)
responseTrace	internal variable that controls what trace this task will use to save out subject responses (usually you will not set this)
phaseTrace	internal variable that controls what trace this task will use to save out the phase number (usually you will not set this)
parameterCode	For parameters that have groups
private	A parameter that you can do whatever you want with
randVars	random variables
fudgeLastVolume	When you synchToVol or keep time in volumes, and want to have the experiment run for a set number of trials, the experiment won't usually end because in the last segment it is waiting for a volume to come in that never will. If you set this to 1, it will fudge that last one so that the experiment ends one TR after the last volume is aquired.

updateTask

purpose: updates the task
usage: [task myscreen phaseNum] = updateTask(task,myscreen,phaseNum)

argument	value
task	task variable, note task **must** be a cell array. If you only have one task phase, make phaseNum=1 and task a cell array of length one.
myscreen	myscreen variable returned by initScreen
phaseNum	The task phase you are currently updating. If you only have one phase, set to 1, for multiple phases, update Task will take care of switching from one phase to the next.

tickScreen

purpose: updates the screen
usage: [myscreen task] = tickScreen(myscreen,task);

argument	value
myscreen	myscreen variable returned by initScreen
task	task variable

This function calls mglFlush to update the screen when it is needed and also checks for volumes and keys etc. Called with in main loop.

upDownStaircase

Implements a staircase for control of stimulus variable values. Type 'help upDownStaircase' for details. Also see taskTemplateFlashingStaircase.m and taskTemplateStaticStaircase.m for examples of using this function.

jumpSegment

Allows you to force a move move to the next segment or the next trial:

```
task = jumpSegment(task) % this will end the segment and move to the next one
```

```
task = jumpSegment(task,inf) % this will end the trial and start a new trial
```

Sample programs can be found in the /Task subfolder, and include:

- **taskTemplate** (Stripped down version that contains only the functions you need to write a task. This is a good starting place for any new experiment you want to program).
- **taskTemplateContrast10bit** An example of how to implement 10 bit contrast (instead of 8 bit).
- **taskTemplateDualMain** and **taskTemplateDualSubsidiary** These provide an example of how to program a dual task.
- **taskTemplateFlashingStaircase** An example of implementing a staircase with flashing stimuli.
- **taskTemplateStaticStaircase** An example of implementing a staircase with static stimuli. This is useful if you want to keep track of reaction times at faster than the screen update.
- **taskTemplateReactionTime** Shows you how to keep reaction times with better precision than the screen update. Note that you cannot update your stimuli while you are waiting for a subject response.
- **testExperiment** An example program with moving dots.

When things don't work, there are some very simple bugs you should check for before losing hope.

Return variables

Many of the routines return multiple values–it is **really** important that you receive these properly. For example, initTask returns both the task **and** myscreen. Make sure that you call it as follows:

```
[task{1} myscreen] = initTask(task{1},myscreen,@startSegmentCallback,@screenUpdateCallback,@responseCa
```

If you did not have the myscreen in the left hand side, then initTask will not set fields (primarly myscreen.stimtrace) correctly in your myscreen–which can make saving out traces incorrect. Look out for this not only in initTask but in any function like updateTask to make sure you are always setting the appropriate return variables correctly (check the help on the function–or model your calls after the ones in the templates).

Screen flashes

Sometimes the screen might appear to flash momentarily with your stimulus in a way you don't expect. Often this is due to not calling mglClearScreen at the appropriate place (like at the beginning of a segment or at the beginning of the screenUpdateCallback). The reason for this occurring is often because you are drawing to the back buffer of the double buffered screen and it already has something from before when it was being displayed. Clearing the screen as you start drawing will insure that you don't have any junk from the last screen draw that will show up.

Matlab control

Don't let the window from which you are calling your code be on the screen that is taken over by MGL, or you will lose the ability to stop the code from running or exit if your code is interupted by an error.

Clear all

Mgl keeps the status of the display in the global variable called MGL. You can look at the values set in MGL by doing:

```
global MGL MGL
```

Be aware that if you clear the MGL variable then mgl will no longer know the status of the screen. The current version handles this by closing the display if you clear the MGL variable. If for some reason, you get stuck with an open display that you cannot close, you can try to call mglOpen again and then close. If you have reset the gamma table, you can go into your System Preferences/Displays/Color and reset the gamma table back to normal there. On Linux, you can use xgamma to set the gamma table, or easier, use NVIDIA's settings manager or the ATI settings manager to restore gamma tables. If you still cannot close the display, you can always do option-open apple-escape and quit out of matlab (on Mac). On Linux, Ctrl-C may work, or you may have to kill the matlab process from a terminal. If you only have one screen and it is locked, you can type Ctrl+Alt+F5 to drop out of the X server temporarily; this will allow you to login and kill the Matlab process without restarting X (Ctrl+Alt+F7 to return to X).

Spelling, capitalization, and periods

If an important variable (e.g. stimulus) is misspelled (e.g. sitmulus), or a variable or function name mis-capitalized (e.g. inittask instead of initTask), or you forget to put the period between a variable name and a field name (e.g. stimuluscontrast instead of stimulus.contrast) things won't work and it'll be very confusing! MGL will check for some things that it knows about, but it can't know how you've named your variables…

Your task doesn't exit when it's over

First, make sure you've specified a certain number of trials, otherwise it'll go on forever.

Then, make sure you have a while loop around the updateTask call checking the phase number. Even if you only have one phase, you still need to give updateTask a phaseNum variable as input, because the code will only end when the phaseNum gets too big for the while loop (and updateTask increases the phaseNum after all the trials have been run)

Set the data directory

So as to always know where your data will be saved, it's good to set the data directory by doing:

```
myscreen.datadir = datadirname;
```

where datadirname is the full path of the directory to which you'd like the data to be saved.

Make sure your displays aren't mirrored.

Sometimes nothing at all will happen. This can be because your screens are mirrored.

Some common error messages that have to do with cell arrays and how to fix them

Error message saying you haven't specified task.segmin and task.segmax

Make sure you've called initTask with the exact task and phase. For example:

```
task{1}.seglen = 1;
task{1} = initTask(task{1},myscreen,@startSegmentCallback,@screenUpdateCallback,@responseCallback);
```

or

```
task{1}{1}.seglen = 1;
task{1}{1} = initTask(task{1}{1},myscreen,@startSegmentCallback,@screenUpdateCallback,@responseCallbac
```

Error message saying 'Cell contents reference from a non-cell array object,'

Make sure you are calling updateTask with a cell array. This can be confusing, because even if you only have one task and only one phase, you still need to define task{1} (rather than just 'task') and then call updateTask with 'task' - e.g.:

```
task{1}.seglen = 1;
task{1} = initTask(task{1},myscreen,@startSegmentCallback,@screenUpdateCallback,@responseCallback);
```

```
phaseNum = 1;
while (phaseNum <= length(task)) && ~myscreen.userHitEsc
 [task myscreen phaseNum] = updateTask(task,myscreen,phaseNum);
end
```

If you're running two tasks, and need to differentiate them, then even if they each only have one phase, you must define task{1}{1} and task{2}{1} and then call updateTask with task{1} and task{2}. For example:

```
task{1}{1}.seglen = 1;
task{1}{1} = initTask(task{1}{1},myscreen,@startSegmentCallback,@screenUpdateCallback,@responseCallbac
task{2} = setSecondTask(myscreen); % as long as setSecondTask returns a cell array
```

```
phaseNum = 1;
while (phaseNum <= length(task{1})) && ~myscreen.userHitEsc
 [task{1} myscreen phaseNum] = updateTask(task{1},myscreen,phaseNum);
 [task{2} myscreen] = updateTask(task{2},myscreen,1);
end
```

Error message saying 'Attempt to reference field of non-structure array,'

Make sure that you call updateTask and also return from updateTask with the same cell array (see above).

Error message 'Input argument "tnum" is undefined,'

Make sure you are passing in a phaseNum argument in your updateTask function call.

Problem starting/stopping the eye tracker

To control the eye tracker you must have the file

```
mgl/task/utils/readDigPort/writeDigPort.c
```

mex'd. You can do

```
cd mgl/task/utils/readDigPort
mex writeDigPort.c
```

This compiles the program to send the digital pulse to the eye tracker. Also, if your task crashes at the beginning after it prints out

```
Dev1/port2
```

or something similar to that, you probably just need to recompile writeDigPort.c. If you continue to have problems and want to give up, you can delete the mexfile writeDigPort.mexmac and then mgl won't try and set the digital port (This is the default condition that the mgl library is in, because most computers don't have the NI card installed).

Contributors

Design and Implementation

- Justin Gardner (Mac OS X, OpenGL, sample programs, documentation, mgl/utils and mgl/task)
- Jonas Larsson (Linux, OpenGL, Mac OS X and mgl/utils)

Other contributors

- Denis Schluppeck (OpenGL and documentation)
- Shani Offen (sample programs and documentation)
- Franco Pestilli (sample programs and documentation)
- Eli Merriam (OpenGL)
- Taosheng Liu (Windows alpha)
- Christopher Broussard (Mac OS X)

'); document.write(''); function installJsMath() { jsMath.Process(document); } addInitEvent(installJsMath);