Goal

The goal of this lab is fairly straightforward. I would like to you make some initial progress on programming a "feature search" experiment.

In doing so, you will be making use of

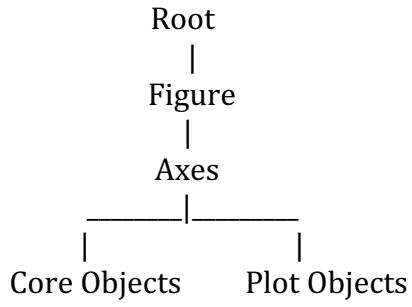
- your previous knowledge of MATLAB,
- your understanding of good experimental design, and
- a number of small programs I've written that will enable you to more easily use MATLAB's graphics capabilities.

An Introduction to Graphics Programming in MATLAB

Visual search experiments involve visual displays. One way to create these displays is with computer graphics.

The Hierarchy of Graphics Objects in MATLAB

The graphics objects we will be concerned with are organized in MATLAB according to the following hierarchy.



What this means is that graphics are composed in **figure** windows, which are attached to the **root**. Figures have internal coordinates which may optionally be set and displayed using **axes**. These coordinates are used to position simple **core objects** and more complex **plot objects** that are built up out of core objects.

Graphics Handles

All graphics objects are referenced by **graphics handles**. A handle is actually a structure which contains a lot of information about the graphics object to which it refers. The handle of the root is always simply the number 0. Try the following:

```
>> hfig1 = figure;          % Creates a figure window
                            % and places its handle in the variable hfig1.

>> hfig1                    % Displays the handle that was just created.

>> get(0, 'CurrentFigure')  % Returns the handle of the current figure, if the
                            % current figure exists.
```

Properties of Graphics Objects

While handles contain some information about a graphics object, they do not contain all the information about them. Different types of graphics objects have different collections of properties. To learn the current value of a *particular* property of a graphics object, use the **get** function, which has the following syntax:

get(*object_handle*, '*PropertyName*')

This is what we just did to confirm the handle of the figure we just created.

It is sometimes useful to see *all* the properties an object has, and their current values. This is done using the syntax:

get(*object_handle*)

Use the **get** function to find out what properties the *root* has.
Then use the **get** function to find out what properties the *figure* we created has.

(There's no need to record this information.)

You'll notice that one of the properties that the figure has is *Color*, and it is currently set to the default value

Color: [0.9400 0.9400 0.9400]

The three numbers in this array are a recipe for creating a color from a combination of the primary colors

[*Red*, *Green*, *Blue*].

Each number represents the fraction of the available mount of that color used.

To change the value of a property, use the **set** function. The syntax is:

set(*object_handle*, '*PropertyName*', *PropertyValue*)

where the *PropertyValue* must be appropriate for the named property.

For example, *Color* must be assigned a vector of three numbers, each of which is between 0 and 1 (inclusive).

Try each of the following, predicting what will happen to the background color of the figure window before you execute the function:

>> **set**(hfig1, 'Color', [1 0 0])

>> **set**(hfig1, 'Color', [0 1 0])

>> **set**(hfig1, 'Color', [0 0 1])

Continue to experiment, setting different values to Color until you have a background color you like. Record the value you settle on below, and describe the color it produces.

Axes

Axes are used to locate graphics objects in a figure. Use the **axes** function now to create some axes:

>> hax1 = **axes**; % Creates axes and places the handle to them in variable *hax1.*

What happened to the figure when you did this?

By default, the current figure is assigned as the *Parent* of these axes if no other *Parent* has been specified. Use the **get** function to confirm this:

>> **get**(hax1, 'Parent')

Now use the **get** function to find out all the properties these axes have.

>> **get**(hax1)

There are, again, many properties, and there's no need to record all this information. Note, however, that unless you explicitly modify these properties they are set to default values you see.

Among these properties, you will see two that determine the minimum and maximum values on each of the axes for the figure:

XLim : [0 1]
Ylim : [0 1]

Change both axis limits now to [0 10] using the set function:

>> **set**(hax1, 'XLim', [0 10])
>> **set**(hax1, 'YLim', [0 10])

How did the figure window change?

Another property in the list of axes properties will be of interest to us:

Visible: 'on'

This indicates that the axes not only exist, but are being explicitly displayed in the figure window.

Since we're just using axes to help us position objects in the figure, we'll eventually set this property to 'off'.

Use the **set** function to make the axes disappear from the figure window. Then make them reappear.

>> **set**(hax1, 'Visible', 'off')

What happened in the figure window?

>> **set**(hax1, 'Visible', 'on')

Again, what happened?

Patches

Now we'll put some more interesting objects in the figure. The **patch** function is used for displaying one or more polygons arranged in a specified way. We'll use it here in its simplest form, to create a single filled polygon. The syntax is:

patch(*X*, *Y*, *C*)

where X is a vector of the x-coordinates of the points of the polygon,
Y is a vector of the y-coordinates of the points of the polygon, and
C is an RGB vector specifying the fill color.

Try the following:

```
>>  X = [ 0  1  0.5 ];
>>  Y = [ 0  0  1];
>>  C = [ 1 0 0 ];
>>  hpat1 = patch( X, Y, C );
```

What do you see in your figure now?

Adjust your fill color so that it contrasts with the background color around of the border of the figure window, then make the axes invisible. See what happens, then make the axes visible again.

```
>> set( hax1, 'Visible', 'off')
```

What happened?

```
>> set( hax1, 'Visible', 'on')
```

What if you wanted to create another triangle in a different location with the same shape and orientation as your first one? If you want the lower left-hand corner of the new triangle to have coordinates (2, 3), for example, do this:

```
>> newX = X + 2
>> newY = Y + 3
>> hpat2 = patch(newX, newY, C);
```

Now use the **get** function to see what properties these objects have.

Use the **set** function to change the *FaceColor* of the first triangle.

Then use the **set** function change the location of the second triangle *without changing its shape*!

Finall, use the **set** function to make the axes invisible.

CHECK POINT: Call me over to show me the changes you've made to your figure.

You've learned something now about how to use graphics functions and s to create a graphics figure in MATLAB.

Go to the figure window now at save it to your G: drive as *Lab7_FirstFigure*.

Working With an Existing Graphics Figure

Everything we've done so far has been predicated on knowing the graphics handles for the figure and everything in it. What can we do if we're given an existing figure, one that we didn't create, and want to modify it?

I've created the figure *FigureExample.fig* and placed it in our MATLAB folder on the S: drive. Use the **hgload** function to load this figure and store its handle:

>>hfig2 = **hgload**('FigureExample');

Note that this figure opens up in a new window. Leave both figure windows open!

You'll notice that this figure contains three triangles and that no axes are visible. The axes are there, however – they're the only direct "child" of the figure.

Use the **get** function to get the handle of these axes for this figure as follows:

```
>> hax2 = get( hfig2, 'Children' );
```

Confirm that the axes are hidden by using the **get** function:

```
>> get( hax2, 'Visibile' )
```

The triangles that you see are "children" of the hidden axes. Use the **get** function to get their handles, as:

```
>> hax2_children = get( hax2, 'Children' );
```

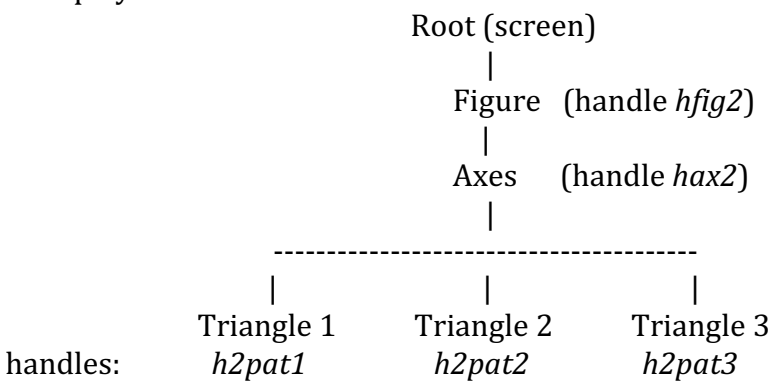
This give you an array of handles, one for each of the triangles. To confirm that you have as many of these handles as triangles, use:

```
>> length(hax2_children)
```

Now assign each of those handles to more convenient variables, such as:

```
>>h2pat1 = hax2_children(1);
>>h2pat2 = hax2_children(2);
>>h2pat3 = hax2_children(3);
```

The following diagram shows the relationship between the various objects in this graphics display



Which triangle is which?
Probably the easiest way to distinguish them is by their color. Try:

```
>> get( h2pat1, 'FaceColor' )
>> get( h2pat2,  'FaceColor' )
>> get( h2pat3, 'FaceColor' )
```

Each one of these will return a vector with three values, corresponding to the "red", "green", and "blue" components of the triangle's color.

Can you tell which triangle is which? Make a note of this:

Another way to distinguish the triangle is by position. Try:

```
>>  get( h2pat1, 'XData' )
>>  get( h2pat2, 'XData' )
>>  get( h2pat3, 'XData' )
```

Once again, each of these commands will return a vector with three values, corresponding to the *x*-coordinates of the three points in the corresponding triangle.

By the way, the scales on the axes range from 0 to 10, as you can learn by:

```
>> get( hax2, 'XLim' )
>> get( hax2, 'YLim' )
```

At this point in the lab, you should be oriented reasonably well to the structure of MATLAB graphics objects and some of the low-level tools for working with them.

Working With Higher Level Graphics Tools

It is convenient to be able to work with graphics tools that accomplish what you want to do in a more natural way. I've created some of these that you can use to create displays for your visual search experiment. They can be found in our MATLAB folder on the S: drive (which you have access to by default).

Hide

Use this function to render all the children of the given axes invisible.

>>**Hide**(hax2)

What happened to the figure? Where did everything go?

(Open the file *Hide.m* and study it to see how this was accomplished.)

Reveal

To get things back to normal, try:

>>**Reveal**(hax2)

(Again, open the file for this function and study it to see how this was accomplished.)

CopyMovePatch

One modification you might want to make to a display is to add more triangles to it. You can do this by *copying* one of the triangles – say the one with handle *h2pat3* – to a new location in the display, as:

>> h2pat4 = **CopyMovePatch**(h2pat3, hax2, 5, 1);

The full syntax for this command is:

[hcopy] = CopyMovePatch(hpatch, h_new_parent, Delta_X, Delta_Y)

where *hcopy* is the handle for the copy,
hpatch is the handle for the original patch object,
h_new_parent is the handle for the parent of the copy, and
(*Delta_X*, *Delta_Y*) is the translation of the copy position from the original patch postion.

To see how this function works, open its file and study it.

The function **CopyMovePatch** is easily modified to make a function that will only move an *existing* triangle rather than create a new one.

Do this now, save it to your own directory as **MovePatch**, and test it out.

CHECK POINT: When you have it working, call me over to check it.

Notes:

There's one more function I'd like you to try:

>>[Response, ResponseTime] = **GetResponse**(hfig2)

After you enter this line, press any key and see what happens.

As the name suggests, this is a function you can use to learn how a person responds to a display as well as the time it takes them to do so.

(Open that file and study it to see what it is doing.)

Other Functions That May Be Useful to You

I've written some other small graphics functions that you may find useful.

Open the files for these and see how they work. Record their syntax and any notes about their performance below:

SetUpFig

RedTriangle

CleanSlate

Feature Search Trial

You're now ready to write a function that will run a single trial of a feature search experiment. It should have the following syntax:

[TargetSeen, ResponseTime] = **FeatureSearchTrial**(N_Distract, TargetPresent)

where

TargetSeen is a logical variable that is TRUE if the subject saw the target,

ResponseTime is the response time,

N_Distract is the number of distractors present in the display, and

TargetPresent is a logical variable that is TRUE if the target is actually in the display.

This function should create a display with *N_Distract* red distractor triangles on a blue background with invisible axes scaled from 0 to 10.

Randomize the triangle locations across the display (using integer coordinates for the lower left corner of the triangles). This MATLAB function will be useful:

randi([0 9]) generates a random integer between 0 and 9.

If *TargetPresent* is TRUE, create a yellow target triangle at a random location in the display with integer coordinates.

You can use **CopyMovePatch** to help you do all this after initializing with **SetUpFig** and **RedTriangle**. Be sure to render the initial triangle invisible afterwards.

Your function should also record the response in *TargetSeen* and the response time in *ResponseTime*. **GetResponse** will help you here.

Interesting Isssues

While maintaining the size and orientation of these triangles, how can you ensure that they do not overlap one another?

How can you ensure that all the objects in the display appear at the same time rather than being (gradually) added to the display?

Programming Advice

1. Open a new **function** file in the MATLAB editor.

2. Put the function syntax line at the top of the file right after the word **function**:

[TargetSeen, ResponseTime] = **FeatureSearchTrial**(N_Distract, TargetPresent)

3. Before you do anything else, write the top comments that give the computational theory for your function. This will essentially be a summary of the description of this function on the preceding page. This should not, however, include details of implementation.

4. Now write out the main steps of your algorithm. I suggest:

% Set up the figure and triangle template.

Hint: Use the functions **SetUpFig** *and* **RedTriangle** *to do this.*

% Make N_Distract copies of the distractor triangle and
% place them at random integer coordinates within the figure.

Hint: Use a **for** *loop and* **CopyMovePatch**.
You'll also need a HandleArray to store the handles for each new distractor you create. Initialize this before the loop using the ***zeros*** *function.*

% If TargetPresent == true, change the color of the triangle template to yellow,
% and then move it to a random integer coordinate within the figure.
% Otherwise, make the triangle template invisible.

Hint: Use the set function to change properties of the triangle, and use your **MovePatch** *function to move it.*

% Record the response and response time.

Hint: Use the function **GetResponse** *to do this.*

5. Implement the algorithm using MATLAB code.
I recommend following the hints above.
TEST each section of code as you write it – don't just write the whole function first. You're more likely to be successful and besides, it's more fun that way!