UNIVERSITY OF CALIFORNIA BERKELEY Engineering 7
Department of Civil and Environmental Engineering
Spring 2013 Professor: S. Govindjee

Basic Graphics and GUIs

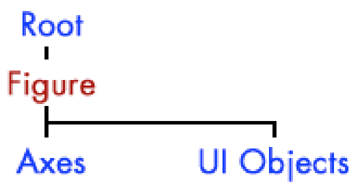
1 Introduction

MATLAB's graphics environment is essentially an object oriented system. All items rendered upon the screen are instances of particular MATLAB objects. Even the figure window within which we see the renderings is an object. As such all graphical operations can be considered as the application of constructors and methods.

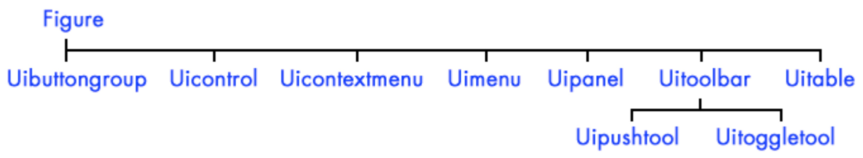
In MATLAB's terminology the basic graphics objects are called "handle graphics objects" and they are the drawing elements used by MATLAB to display data and to create graphical user interfaces (GUIs). The instances of the objects themselves are often simply called handles in MATLAB documentation. Knowing the handle of a graphics object allows one to alter its internal data (properties) via the `set()` method. All MATLAB graphics objects have a `set()` method for setting the properties and a `get()` method for querying the properties.

The basic class hierarchy is shown in Fig. 1. At the top of the hierarchy is the root window. This is actually the computer screen (that you see) when you login to the computer and there can only be one instance of the root object; it is always present and you cannot create or delete it. Any graphical operation that you wish to execute is contained inside the root object; i.e. when you perform a graphical operation you are actually setting some of the data associated with the root object. In the MATLAB environment one (always) first creates a figure object within the root object. All other graphic objects are then created as part of the data of the figure object. Within the figure object one can create 4 different types of objects as shown in Fig. 1. Three of the objects that can be created within the figure object are associated with GUIs (buttons, pulldown menus etc.) – the UIcontrol object, the UImenu object, and the UIcontextmenu object. The fourth object one can create within a figure object, the axes object, is perhaps the most common and is used for the drawing of lines etc. Within this object one can create MATLAB's primitive drawing objects: the Image object, the Light object, the Line object, the Patch object, the Rectangle object, the Surface object, and the Text object. For all of these objects (figures, lines, UImenus, etc.) one can create single instances of them, arrays of them, and combinations of them. For example in Fig. 2, one has an example of a Figure object which contains several different types of objects (some of which are labeled).

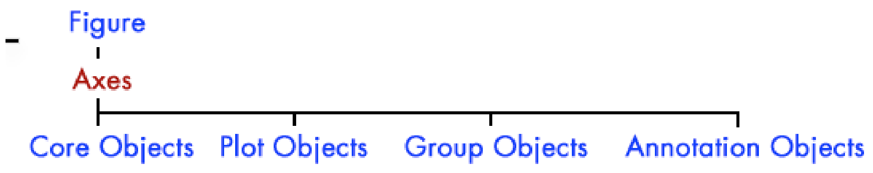
Note that in what has been discussed so far we have made no mention whatsoever of the `plot()` method which you have been using on a regular basis. Within the context of what we have discussed so far, `plot()`, like many other MATLAB rendering functions, represents a method that performs a complex set of tasks which involve the creation and manipulation



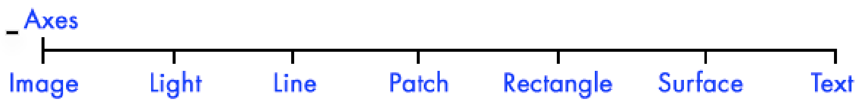
(a) Top level tree.



(b) UI Object subtree.



(c) Axes subtree.



(d) Core Objects subtree.

Figure 1: MATLAB handle graphics class hierarchy.

of the handle graphics objects we have just introduced. As such we will not discuss it or others like it in these brief notes. Instead we will concentrate upon the primitive graphic objects and their manipulation so as to give you a solid feel for the fundamentals associated with MATLAB graphics and graphics environments in general.

2 Figure objects

2.1 Creation and copying

The figure object is always required in MATLAB before any graphical operations can be performed. To create a figure object one uses the figure constructor `figure`. For example:

```
>> h = figure;
```

will create a figure object and store a "handle" to it in `h`. Note that in the workspace window `h` is listed as being of class "double array" instead of class "figure". `h` is a handle (pointer) to an instance of the figure object but not the object itself. MATLAB does not allow the user direct access to the variable representing the object. All access is mediated by the handle. Thus if one types:

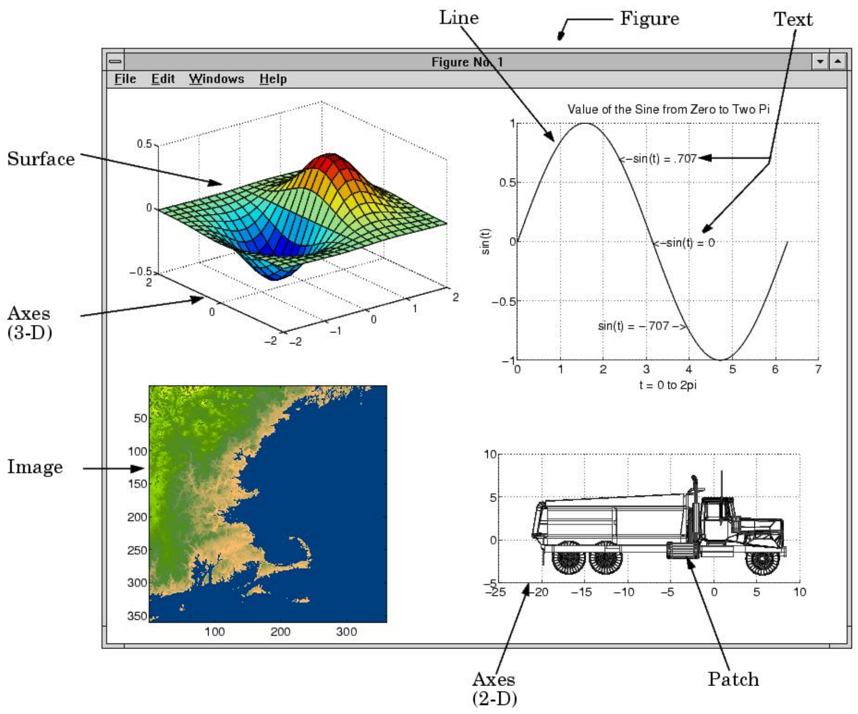


Figure 2: Example Figure object with multiple Axes objects and various other sub-objects as labeled.

```
>> b = h;
```

one will *not* create a second figure object. One will simply be able to reference the original object using either the variable (handle) `b` or `h`. To create a second figure object one needs to re-execute the figure constructor (e.g. `g = figure`) or, alternatively, one can use the `copyobj` method. For example to make a copy of the figure object `h` and all its children one would type the following:

```
>> root_object_handle = get(h,'Parent');
>> g = copyobj(h,root_object_handle);
```

The first line gets the handle to the Parent of `h`, which happens to be the root object and which happens to always be `0` in MATLAB. The second line creates a complete copy of `h` and all its children and places that copy within the data of the root object. `g` is a handle to this new figure object and this new figure now appears in addition to the original figure. Note the `copyobj` method also applies to all handle graphic objects not just the figure object.

2.2 Getting and setting properties

The properties[1] of the figure object can be examined by typing:

```
>> get(h)
```

This will give a complete list of all the properties/data of the figure object. To see the values of just one of the properties of the figure one would type, for example,

```
>> get(h,'Units')
```

Some of the properties can only take values from a fixed set of keywords. To see what properties can be set and what their admissible values are, one can type:

```
>> set(h)
```

Properties which only take values from a fixed set show the set of admissible values in square brackets and the default values are contained within braces. To actually change a property value one would type, for example,

```
>> set(h,'Units','inches')
```

[1]A complete interactive listing of the properties of all MATLAB graphics objects can be found at the MATHWORKS web site. Type `>> doc` and then click on `Handle Graphics: Object Properties` in the center right pane.

2.3 Deleting

Figure objects can be deleted using the `delete()` method. For example to delete a figure object whose handle is `h` type

```
>> delete(h)
```

This syntax also works for deleting all other graphics objects.

3 Axes objects and their children

To render anything within the figure object one needs to create children objects. The possible children are those shown in Fig. 1. In this section we will discuss the axes object and its children. In the subsequent section we will discuss the creation of GUI-type objects.

3.1 Axes object

The axes object places a set of axes within a figure object. The area of the figure which is occupied by the axes is defined in the constructor call (using coordinates normalized to 0 to 1). Further one can set the scaling of the axes in the constructor or later using the set command. Also one typically sets the visibility of the axes to off in the constructor or later by using the set command; the axes are usually only displayed when making graphs. As an example consider the creation of an axes object within a figure object

```
>> h = figure;
>> a = axes('Parent',h,'Position',[0.5 0.5 0.4 0.4]);
```

The first pair in the constructor indicates that the axes object will be a child of the object associated with the figure handle `h`. The second pair indicates that the axes will occupy the region of the figure window defined by `[0.5 0.5 0.4 0.4]` where the order of the quadruplet is [left bottom width height] – all in normalized coordinates. The return argument `a` is a handle (pointer) to the axes object. The axes scaling can be adjusted using the set command; e.g.

```
>> set(a,'XLim',[-1 1],'YLim',[50 60]);
```

Note that this simply changes the scaling of the axes not the region that the axes occupy within the figure object which is associated with the Position property. When rendering objects within the axes one would render them using coordinate positions in the range $\{(-1,1) \times (50,60)\}$; see Fig. 3 for the current state of the figure.

To hide the display of the coordinate axes one can set the visible property of the axes to off

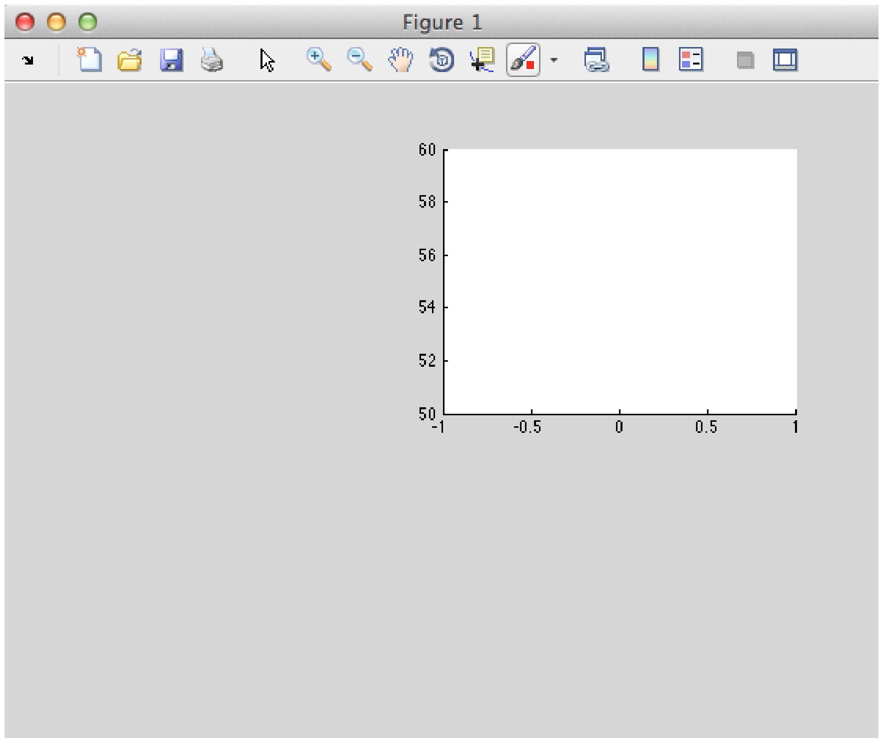


Figure 3: Example figure object containing an axes object at Position [0.5 0.5 0.4 0.4], XLim [-1 1], and YLim [50 60].

```
>> set(a,'Visible','off');
```

The coordinate axes are no longer explicitly displayed but are still there and available for rendering purposes. To re-display them type

```
>> set(a,'Visible','on');
```

To add additional axes to the figure simply re-execute the axes constructor; e.g.

```
>> a2 = axes('Parent',h,'Position',[.3 .3 .4 .4]);
```

This creates a second set of axes with handle `a2` which slightly overlap our first set. Note that the rendering order is the same as the creation order. To re-order the rendering sequence one needs to re-order the Children of the figure object. If you type

```
>> get(h,'Children')
```

you will see that the two axes we have created are the children of the figure object. They are stored in an array in reverse order of their creation; note the rendering is done starting with the end of the array and moving forward to the first – i.e. the last created child. To change the rendering order we could type

```
>> set(h,'Children',[a a2]);
```

and this would cause our first set of axes to appear last and thus on top of the second set of axes.

To see all the properties of an axes object you can use the `get()` method on an axes handle; use `set()` to set the properties in the same fashion as we did with figure objects. As a further example note that the axes objects can be used to plot in 3-D. The range of the z axes is set using the ZLim property and to see it rendered one needs to set the Projection property of the axes; e.g.

```
>> set(a,'ZLim',[0 100],'Projection','perspective')
```

The orientation of the perspective view can be adjusted with the View property which is an array containing the azimuth and elevation of the viewpoint in degrees; try for example

```
>> set(a,'View',[30 30])
```

to produce Fig. 4. Two more convenient way of setting this property are to use the `view()` method or to use the pull down Tools menu and choose the Rotate 3D item; in fact many of the properties of items in a figure can be adjusted using the pull down menus.

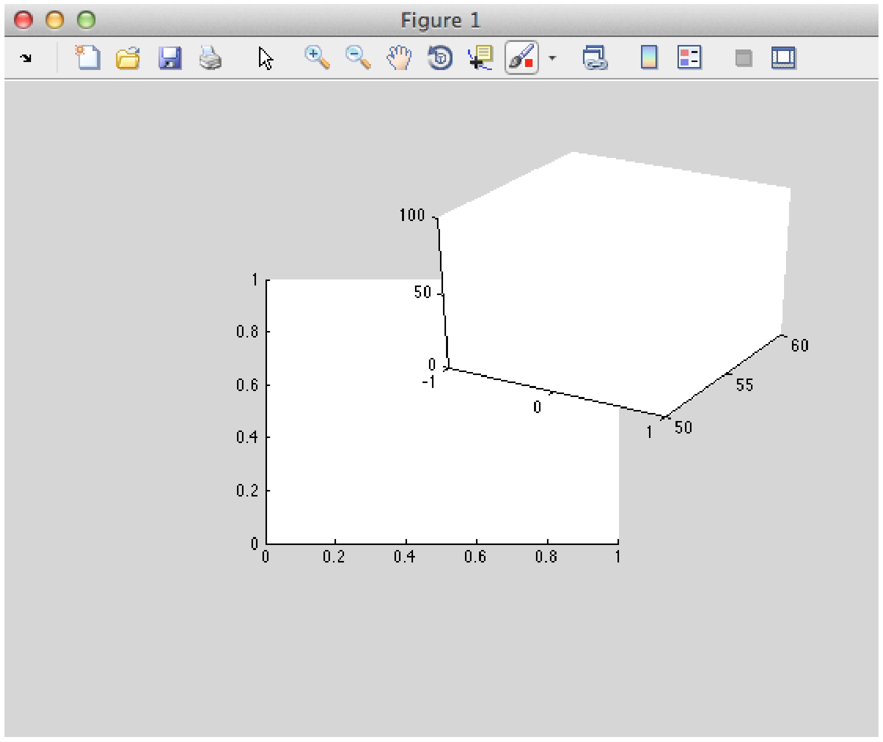


Figure 4: Example figure object containing 2 axes objects: one with the projection property set to the default value of 'orthographic' and the other set to 'perspective'.

3.2 Line object

One of the most basic objects that one can render is a line object. A line object is created using the `line` constructor. The line object itself is a child of a particular axes object and its principal data are the coordinates of the end points of the line. For example,

```
>> l1 = line('Parent',a,'XData',[0 0 0],'YData',[50 55 55],...
             'ZData',[50 50 100]);
>> l2 = line('Parent',a2,'XData',[0.2 0.6],'YData',[0.6 0.2]);
```

will draw a 3D line and a 2D line on our figure from the previous section. The handles to the lines are the return arguments. Note that the second line objects creation actually had an effect on the `a2` axes in that it altered the limits of the axes coordinates. To prevent this from happening one needs to change the XLimMode and YLimMode properties of `a2` from 'auto' to 'manual'.

To see the properties of a line object or to set them use the `get()` and `set()` methods. For example to change the color of the line associated with handle `l1` type

```
>> set(l1,'Color','r')
```

As a second example consider the following function which creates a blue line object and spins it around on the screen. The program uses a bit of Math 54 but you can safely ignore the details of those parts in trying to understand the program.

```
function rotate_blue_line

 % Create the main figure object and axes
 h = figure;
 a = axes('Parent',h,'Position',[.1 .1 .8 .8]);
 set(a,'Projection','perspective','View',[30 30],'Visible','off');
 set(a,'XLimMode','manual','YLimMode','manual','ZLimMode','manual');

 % Create the initial data for the line object
 x0 = [.2 .6 .6 .2 .2 .2 .6 .6 .2 .2];
 y0 = [.2 .2 .6 .6 .2 .2 .2 .6 .6 .2];
 z0 = [.2 .2 .2 .2 .2 .6 .6 .6 .6 .6];

 % Create the line object
 l = line('Parent',a,'XData',x0,'YData',y0,'ZData',z0,'Color','b');

 % Setup the axis of rotation
 pole = [.6; .6; 0];

 % Spin the object from 0 to 50*pi in 500 increments
 for theta = linspace(0,50*pi,500)

   % The rotation matrix from Math 54
```

```
    R = [ cos(theta) sin(theta) 0;
         -sin(theta) cos(theta) 0;
                   0          0  1];

    % Transform the data points of the line object using Math 54 stuff
    for i = 1:length(x0)
     newloc = R*( [x0(i); y0(i); z0(i)] - pole ) + pole;
     x(i) = newloc(1);
     y(i) = newloc(2);
     z(i) = newloc(3);
    end

    % Reset the data for the line object using the rotated
    % coordinates of the X,Y,Z Data properties
    set(l,'XData',x,'YData',y,'ZData',z);

    % Force the immediate drawing of the updated object
    drawnow;

    % Pause for visual effect
    pause(0.1);

  end
```

3.3 Patch object

Another basic graphics primitive is the patch object. A patch is a filled polygon (either 2D or 3D). As with the other objects discussed so far one can use `set()` and `get()` to set and see a patch objects properties. The basic data needed in setting up a patch object are the coordinates of the vertices of the polygon and the color of the polygon. The vertices are stored in arrays and the color is specified by a three element array containing the 'RGB' values of the color; e.g. [.4 .3 .8] would indicate a color that uses 4 parts red 3 parts green and 8 parts blue to create a color close to a light purple (values must be between 0 and 1). For example, type

```
>> g = figure;
>> a = axes('Parent',g,'XLimMode','manual','YLimMode','manual');
>> p = patch('XData',[.2 .4 .5 .6],'YData',[.1 .1 .3 .9]);
>> set(p,'FaceColor',[.4 .3 .8]);
```

Note that in the above we did not set the position of the axes or the ranges of the coordinate axes. If you do not do so, then MATLAB will simply fill in some defaults. Note that patch objects are 3D so one could also set ZData for them.

3.4 Rectangle object

The rectangle object in MATLAB is a strictly 2D object and is a (possibly filled) object with the shape of a rounded rectangle. The basic construction requires a Position within the parent axes object and a curvature of the x and y sides. For example, the following generates a table of rectangle objects with different curvatures:

```
>> g = figure;
>> a = axes('Parent',g,'XLimMode','manual','YLimMode','manual');
>> r1 = rectangle('Parent',a,'Position',[.1 .1 .1 .1],'Curvature',[0 0]);
>> r2 = rectangle('Parent',a,'Position',[.3 .1 .1 .1],'Curvature',[0 .5]);
>> r3 = rectangle('Parent',a,'Position',[.5 .1 .1 .1],'Curvature',[0 1]);
>> r4 = rectangle('Parent',a,'Position',[.1 .3 .1 .1],'Curvature',[.5 .0]);
>> r5 = rectangle('Parent',a,'Position',[.3 .3 .1 .1],'Curvature',[.5 .5]);
>> r6 = rectangle('Parent',a,'Position',[.5 .3 .1 .1],'Curvature',[.5 1]);
>> r7 = rectangle('Parent',a,'Position',[.1 .5 .1 .1],'Curvature',[1 .0]);
>> r8 = rectangle('Parent',a,'Position',[.3 .5 .1 .1],'Curvature',[1 .5]);
>> r9 = rectangle('Parent',a,'Position',[.5 .5 .1 .1],'Curvature',[1 1]);
```

Note that the first Curvature value represents the fraction of the width of the rectangle which is curved and the second curvature value is the fraction of the height of the rectangle which is curved. Both these numbers are between 0 and 1. See `set()` and `get()` for further properties.

3.5 Text object

The text object in MATLAB allows one to render text. The basic properties of text objects are the position of the text in the parent axes and the string of text to be displayed. For example,

```
>> g = figure;
>> a = axes('Parent',g,'XLimMode','manual','YLimMode','manual');
>> t = text('Parent',a,'Position',[.5 .7],'String','sin(10)');
```

will place the string sin(10) at the coordinates (.5,.7) in the axes object `a`. The handle to the text will be `t`. Text can also be created in 3D by specifying a third element to the Position array. See `set()` and `get()` for further properties such as point size and character font.

4 Graphical user interfaces GUIs

The programming of GUIs in the MATLAB environment is achieved by creating special objects as children of the figure object. Here we will discuss two type of objects: the UImenu object, which allows you to create pull down menus, and the UIcontrol object, which allows you to create slider-bars, buttons, etc. The objects can be placed and labeled and their interaction with the mouse can be controlled through the use of the “callback” property.

The notion of a callback property is essential to the understanding of GUI type object. The essence of the callback property is that whenever the mouse activates part of the object then a particular method (function) is executed. Thus the action associated with clicking on GUI objects is completely governed by the callback function call.

Just as with the other low level graphics objects we have discussed the constructors here return handles to the created objects which can then be used in referencing the object with the `set()` and `get()` methods.

4.1 UImenu object

The uimenu object is used to create pull-down menus on the figure object. The figure object by default has a number pull-down menus but one can add more by creating additional uimenu objects. The constructor is `uimenu` and the syntax for usage is very similar to the other graphics objects we have been dealing with. As an example let us first create a menu labeled 'Rectangle'.

```
>> h = figure;
>> u = uimenu('Parent',h,'Label','Rectangle');
```

This creates a menu object in the top menu bar when the figure has "focus" – is on top and is selected. If you click on 'Rectangle' with the mouse, it highlights it but does not do anything since we have not set up the callback for the menu item. Let us go ahead and set it up so that every time one clicks on the menu it will add a random colored rectangle to the figure. To do this we will first create a function that does this random operation and then we will set the callback property of the menu to execute the function. First let us set up the axes

```
>> a = axes('Parent',h,'Visible','off');
>> set(a,'XLimMode','manual','YLimMode','manual');
```

Now set up the function to make randomly colored random sized rectangles on a given set of axes.

```
function random_rectangle(a)
%
% Input: a -- axes handle to a set of axes with limits (0,1) x (0,1)
% Output: Randomly colored and sized rectangle added to the children of a

r = rand;
g = rand;
b = rand;

x = rand;
y = rand;
w = rand*(1-x);
```

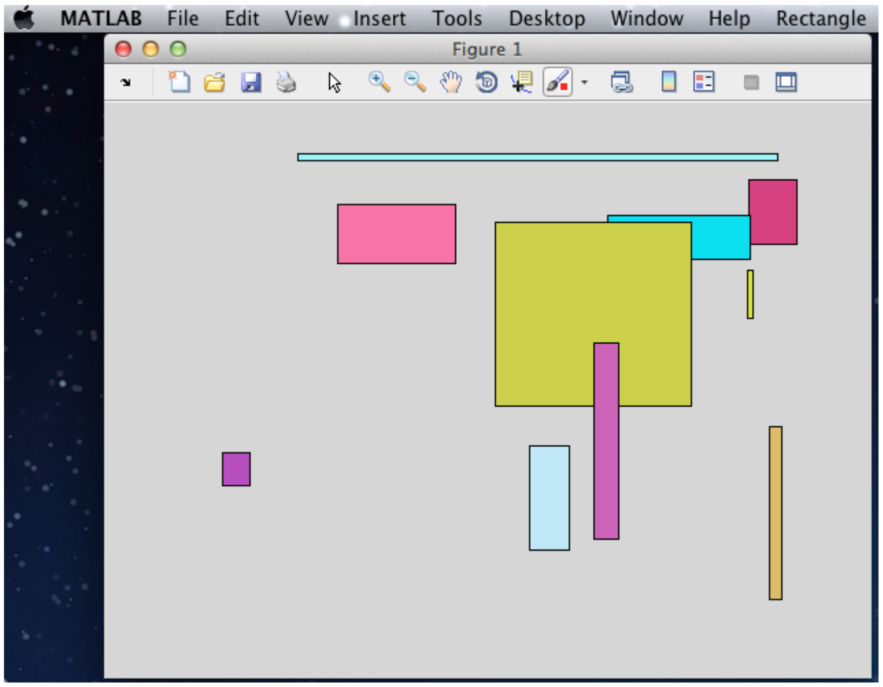


Figure 5: Screenshot of our figure object after clicking 10 times on the Rectangle menu object.

```
h = rand*(1-y);

rectangle('Parent',a,...
                     'Position',[x y w h],...
                     'Curvature',[0 0],...
                     'FaceColor',[r g b]);
```

The only remaining thing to do is to set the callback feature of the uimenu object. This is done by setting the Callback property. This is a string which is executed in the workspace environment. So in our case we can set the callback simply as

```
>> set(u,'Callback','random_rectangle(a)');
```

Now if we click on the Rectangle menu item it will execute the callback string just as if we had typed it at the MATLAB command prompt. Thus for each click, we will see a random rectangle generated in the figure. Fig. 5 show an example screen shot after clicking 10 times.

4.1.1 Submenus

To create submenus we can add other uimenu objects as children of a uimenu. For example to add 2 submenus to our existing uimenu object we can type.

```
>> us1 = uimenu('Parent',u,'Label','Sub menu item 1')
>> us2 = uimenu('Parent',u,'Label','Sub menu item 2')
```

Submenus of submenus are also possible. For each menu item one needs to define the callback property of the menu item to have any functionality. If the menu item is not to have any functionality then the Callback string should be set blank; i.e. to '' (which happens to be the default). This point is important when creating submenus as one typically will not want to have the parent menu execute a callback.

4.2 UIcontrol objects

UIcontrol objects function very similarly to UImenu objects. They only differ in that they need to be positioned within the figure object and their style needs to be defined. The Style property for the uicontrol objects are take from the following list: pushbutton, togglebutton, radiobutton, checkbox, edit, frame, slider, text, listbox, and popupmenu. The labeling of uicontrol objects is done with the String property. Functionality is again determined by setting the Callback property of the object. Fig. 6 show a screen shot with the first 8 of these styles, which were created with the following script:

```
h = figure;

pb = uicontrol('Parent',h,'Style','pushbutton',...
    'Units','normalized',...
    'Position',[0.1 0.8 0.3 0.1],...
    'String','Push Button');

tb = uicontrol('Parent',h,'Style','togglebutton',...
    'Units','normalized',...
    'Position',[0.1 0.65 0.3 0.1],...
    'String','Toggle Button');

rb = uicontrol('Parent',h,'Style','radiobutton',...
    'Units','normalized',...
    'Position',[0.1 0.50 0.3 0.1],...
    'String','Radio Button');

cb = uicontrol('Parent',h,'Style','checkbox',...
    'Units','normalized',...
    'Position',[0.1 0.35 0.3 0.1],...
    'String','Check Box');

eb = uicontrol('Parent',h,'Style','edit',...
    'Units','normalized',...
    'Position',[0.1 0.20 0.3 0.1],...
    'String','Edit Default String');

fb = uicontrol('Parent',h,'Style','frame',...
```

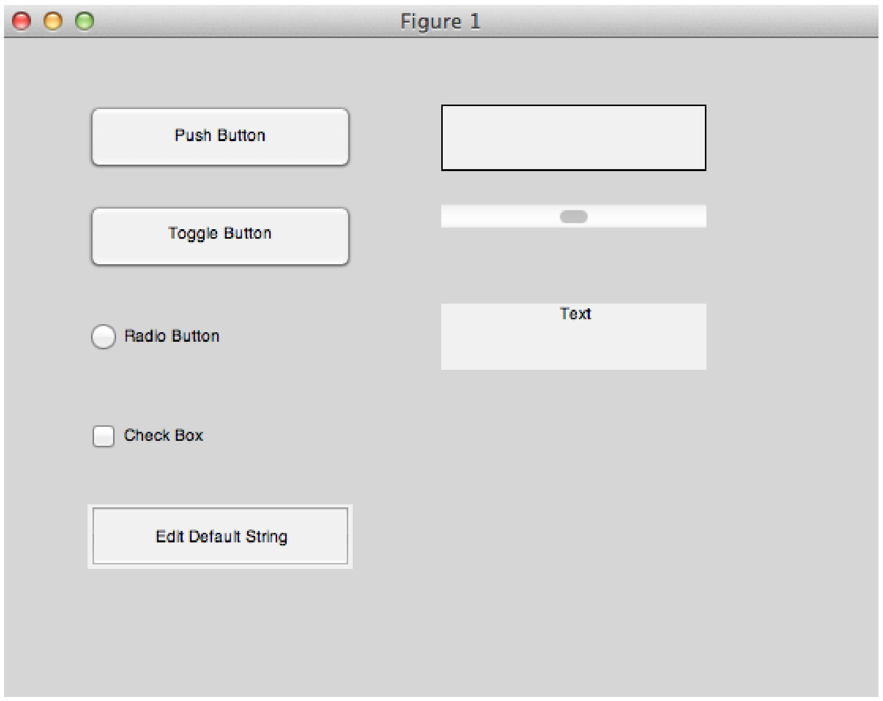


Figure 6: Eight of the unicontrol options.

```
    'Units','normalized',...
    'Position',[0.5 0.8 0.3 0.1]);

sb = uicontrol('Parent',h,'Style','slider',...
    'Units','normalized',...
    'Position',[0.5 0.65 0.3 0.1],...
    'Value',0.5);

tf = uicontrol('Parent',h,'Style','text',...
    'Units','normalized',...
    'Position',[0.5 0.5 0.3 0.1],...
    'String','Text');
```

The setting up of callbacks for the different objects is just as with the UImenu objects. The additional functionality of these objects however is that they can have values. Thus if one has a handle to a UIcontrol object of Style slider, then one can determine the value of the slider by applying the `get()` method on the handle to query the Value property of the slider. For the edit style uicontrol objects, the string contained in the box will be stored as the String property. For radio, toggle, and checkbox buttons, when selected the Value property will be equal to the Max property (default 1) and when not selected the Value property will be equal to the Min property (default 0).

As an example, let us create a program that displays a colored rectangle object of variable color and shape. We will have 3 slider objects to control the color, 3 text objects to display the values of the sliders, and 2 radio buttons to switch between rectangles and ellipses. We will setup our program with an initialization routine `ColorRect` followed by separate callback functions for the UIcontrol objects.

The function `ColorRect` (see below) initializes the figure window and initializes the axes and a default rectangle to be plotted on the axes. It also sets up the 3 sliders, the text displaying the values of the sliders, and the 2 radio buttons for selecting the plotted shape. As much as possible this part of the code uses default values and only sets those values (properties) that are needed in the initialization. For illustration purposes we show two types of callback declarations. With the radio buttons we use string to be executed as we had before. Note, nonetheless, the use of double quotation marks here since the command itself contains a string; e.g. in the second radio button we use the string `'ShapeControl(''ellipse'')'` in the callback declaration because we wish to pass the string `'ellipse'` to the function `ShapeControl`. The second callback style, used with the sliders, employs a function handle: `@ColorSlider`. In this case, MATLAB automatically assumes that the callback function header is of the form `function ColorSlider(hObj,Event)`, where `hObj` is a handle to the object that caused the callback and `Event` is a structure that some UIcontrol object optionally generate with further information about the event that caused the callback to occur. Note that it is possible to have additional arguments to a function handle callback declaration; see the MATLAB documentation for further details.

```
function ColorRect
% Usage: ColorRect
% Purpose: Set up a simple GUI
%
% Output: Initialized GUI

   % Set up the primary objects first the figure, axes, and rectangle
   h = figure;

   a = axes('Parent',h,'Position',[0 .5 1 .5],'Visible','off');

   re = rectangle('Parent',a,'Position',[0 0 1 1],...
                            'Curvature',[0 0],...
                            'FaceColor',[.5 .5 .5],...
                            'Tag','my_re');

   % Set up the uicontrols with all the proper initial values
   s_red = uicontrol('Parent',h,'Style','slider',...
                               'Units','normalized',...
                               'Position',[0 .4 .5 .09],...
                               'SliderStep',[0.03 0.10],...
                               'Value',0.5,...
                               'Tag','my_s_red',...
                               'Callback',@ColorSlider);
   s_red_text = uicontrol('Parent',h,'Style','text',...
                               'Units','normalized',...
                               'Position',[.55 .4 .5 .09],...
```

```
                             'String','Red 0.5',...
                             'FontWeight','bold',...
                             'FontSize',30,...
                             'Tag','my_red_text');

s_green = uicontrol('Parent',h,'Style','slider',...
                             'Units','normalized',...
                             'Position',[0 .3 .5 .09],...
                             'SliderStep',[0.03 0.10],...
                             'Value',0.5,...
                             'Tag','my_s_green',...
                             'Callback',@ColorSlider);
s_green_text = uicontrol('Parent',h,'Style','text',...
                             'Units','normalized',...
                             'Position',[.55 .3 .5 .09],...
                             'String','Green 0.5',...
                             'FontWeight','bold',...
                             'FontSize',30,...
                             'Tag','my_green_text');

s_blue = uicontrol('Parent',h,'Style','slider',...
                             'Units','normalized',...
                             'Position',[0 .2 .5 .09],...
                             'SliderStep',[0.03 0.10],...
                             'Value',0.5,...
                             'Tag','my_s_blue',...
                             'Callback',@ColorSlider);
s_blue_text = uicontrol('Parent',h,'Style','text',...
                             'Units','normalized',...
                             'Position',[.55 .2 .5 .09],...
                             'String','Blue 0.5',...
                             'FontWeight','bold',...
                             'FontSize',30,...
                             'Tag','my_blue_text');



radio_r = uicontrol('Parent',h,'Style','radiobutton',...
                             'Units','normalized',...
                             'Position',[0 .1 .4 .09],...
                             'String','Rectangle',...
                             'SliderStep',[0.05 0.10],...
                             'Value',1,...
                             'FontWeight','bold',...
                             'FontSize',20,...
                             'Tag','my_radio_r',...
                             'Callback','ShapeControl(''rectangle'')');
radio_e = uicontrol('Parent',h,'Style','radiobutton',...
                             'Units','normalized',...
                             'Position',[.55 .1 .4 .09],...
                             'String','Ellipse',...
```

```
                                        'SliderStep',[0.05 0.10],...
                                        'Value',0,...
                                        'FontWeight','bold',...
                                        'FontSize',20,...
                                        'Tag','my_radio_e',...
                                        'Callback','ShapeControl(''ellipse'')');
end
```

The callback ShapeControl deals with what happens when we push the radio buttons. This function is executed due to the callbacks which we set up already in the initialization. The first part used the `findobj` method to retrieve the graphics handles to various objects in our GUI. This is done by using the `Tag` properties that we (uniquely) set for each object in our GUI when we call the UIcontrol constructor. We can query MATLAB using these `Tag` strings and it will return the handles. Thus one does not need to pass the handles in and out of the callback function header. The second part of function toggles the values of the radio buttons and sets the curvature of the rectangle to the appropriate values.

```
function ShapeControl(arg)
% Usage: ShapeControl(arg)
% Purpose: Callback function for adjusting shape
% Inputs: arg -- string to select shape
% Outputs: Update to GUI, set radiobuttons, change plot

% Find handles to radio buttons and rectangle using the Tag values

re      = findobj('Tag','my_re');
radio_r = findobj('Tag','my_radio_r');
radio_e = findobj('Tag','my_radio_e');

% Set the depressed button and turn the other off, set the proper
% curvature of the rectangle
if strcmp(arg,'ellipse')
    set(radio_r,'Value',0);
    set(radio_e,'Value',1);
    set(re,'Curvature',[1 1]);
else
    set(radio_r,'Value',1);
    set(radio_e,'Value',0);
    set(re,'Curvature',[0 0]);
end

end
```

The callback function `ColorSlider` deals with changes in the slider values. Whenever a slider value is changed, the callback function `ColorSlider` is called due to the way in which we set up the callbacks. As with `ShapeControl`, the function first retrieves the graphics handles which it will need. Then it queries the sliders for their values. Next, it uses the input argument `hObj` to determine which slider was actually changed. Next it changes the text associated with the slider that was moved and lastly it set the face color of the rectangle object.

```
function ColorSlider(hObj,Event)
% Usage: ColorSlider(hObj,Event)
% Purpose: Callback to process a changing slider value
% Inputs: hObj -- handle to slider that was changed
%         Event -- event structure (non-empty for select UIcontrols)
% Output: Updated text boxes and rectangle color

   % First grab the textbox object handles using the findobj method
   % as well as the handle to the rectangle and sliders
   re          = findobj('Tag','my_re');

   s_red       = findobj('Tag','my_s_red');
   s_green     = findobj('Tag','my_s_green');
   s_blue      = findobj('Tag','my_s_blue');

   s_red_text = findobj('Tag','my_red_text');
   s_green_text = findobj('Tag','my_green_text');
   s_blue_text = findobj('Tag','my_blue_text');

   % Get the slider values
   r = get(s_red,'Value');
   g = get(s_green,'Value');
   b = get(s_blue,'Value');

   % Determine the tag of the slider that was changed
   slider_tag = get(hObj,'Tag');

   % Change the corresponding text values
   if strcmp(slider_tag,'my_s_red')
     txtstring = sprintf('Red %3.2f',r);
     set(s_red_text,'String',txtstring);
   elseif strcmp(slider_tag,'my_s_green')
     txtstring = sprintf('Green %3.2f',g);
     set(s_green_text,'String',txtstring);
   else
     txtstring = sprintf('Blue %3.2f',b);
     set(s_blue_text,'String',txtstring);
   end

   % Set the rectangle facecolor
   set(re,'FaceColor',[r g b]);

end
```

To start up the GUI program, type

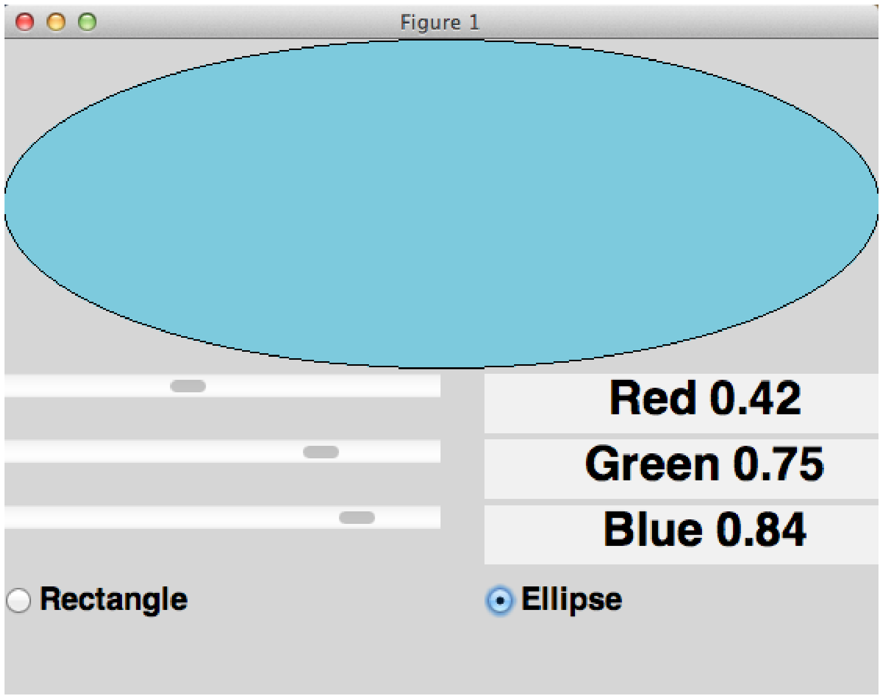


Figure 7: Example screenshot from the `ColorRect` program.

```
>> ColorRect
```

Now move the sliders around and push the radio buttons to see what happens. A sample screen shot is shown in Fig. 7.