User interface basics

Vizard provides a handful of 2D graphical user interface (GUI) options to help you gather data from users. These GUIs are 2D or 'glued to the screen', so they're best used in worlds run in desktop modes or as introductory screens. Vizard has basic GUI elements like sliders, drop-down menus, and a handful of different buttons. These basic elements can be grouped together using the `vizinfo`, `vizconfig`, and `vizdlg` libraries or placed within pop-down menus using `vizmenu`. For a selection of standard prompts and Windows style dialog boxes take a look at `vizinput`. Vizard also supports gathering data from HTML forms through the `vizhtml` library.

In addition to these traditional 2D GUIs, Vizard also allows you to move from 2D selection to 3D selection of objects using screen to world picking and tool tips.

Basic GUI elements

The basic GUI elements are buttons, radio buttons, checkboxes, text boxes, slider bars, progress bars, and droplists. Check out the section on basic GUI elements for more details on the distinctions between these. You can get data from GUI elements either by accessing them spontaneously or by using event callbacks.

```
#Add a progress bar.

bar = viz.addProgressBar('How much?')

#Place it in the middle of the window.

bar.setPosition(.5,.5)

#Get its data.

bar.get()
```

Organizing groups of GUI elements and setting up pop-down menus

If you've got multiple GUI elements within a world, you have several choices for organizing them. A `vizinfo` object creates a panel and adds them in a list from top to bottom. `Vizmenu` allows you to place them in pop-down menus and `vizconfig` creates a tabbed panel for their display. Advanced layouts of GUI elements are supported through the `vizdlg` library. You can also change the appearance of GUI elements or groups of GUI elements by changing their fonts or color schemes

```
#Add the vizinfo panel with variety of GUIs.

import vizinfo

info = vizinfo.InfoPanel('Give me feedback!')

button = info.addLabelItem('Push me',viz.addButton())

slider = info.addLabelItem('Slide me',viz.addSlider())

checkbox = info.addLabelItem('Check me',viz.addCheckbox())
```

Prompts and dialogue boxes

The `vizinput` library has a set of standard dialogue boxes that you can use as prompts.

```
import vizinput



#prompt for a string

userName = vizinput.input('Enter your name:')

print(userName )
```

HTML forms

HTML forms are commonly used to pass data from a browser to a web server. Data is entered through input fields (e.g. text fields, check boxes, radio buttons) and submitted with a button click. The `vizhtml` module supports collecting data from HTML forms displayed in the Vizard graphics window and within a remote browser on the local network.

GUI elements

The basic GUI elements are:

- **Check boxes**
- **Buttons**
- **Label buttons**
- **Radio buttons**
- **Sliders**
- **Droplists**
- **Text boxes**

These elements can exist on their own or they can be grouped together as a vizinfo object or a vizmenu pop-down menu. In general, you can place a GUI element where you want it on the screen using `<GUI element>.setPosition(<coordinates>)`, where the `<coordinates>` are window coordinates going from 0 to 1 and (0,0) is at the lefthand bottom corner of the screen. To set the state of a GUI element, use <GUI element>.set(<valueCheck out the section on getting data from GUI elements for details about accessing the input in your script. For a list of the commands you can use with GUI elements, check out the GUI command table.

Buttons and checkboxes

Buttons and checkboxes are just what they sound like-- onscreen buttons and checkboxes. The main difference between them is that buttons immediately bounce back to the up state after they're clicked while checkboxes remain selected. Radio buttons are a specific kind of button that come in groups where only one radio button in a given group can be selected at one time. Label buttons are buttons with text on them. To change the text on a label button, use `<button>.message(<string>)`.

```
myButton = viz.addButton() #Add a button.

myButton.setPosition(.5,.8) #Set its position.
```

```
myButton.setScale(3,3) #Scale it.



myRadio1 = viz.addRadioButton(1) #Add a radio button to group 1.

myRadio2 = viz.addRadioButton(1) #Add another one.

myRadio1.setPosition(.45,.5) #Set their positions.

myRadio2.setPosition(.55,.5)
```

Sliders and progress bars

Sliders consist of a bar and a tick that slides along it. You can change the size of the tick, the size of the bar, the margin from which it can go to the edge, and the textures applied to its tick and slider, and the orientation of the bar. Progress bars are similar, but they contain text and have a shaded area that slides along the bar instead of a tick.

```
mySlider = viz.addSlider() #Add a slider.

mySlider.vertical() #Orient it vertically.

mySlider.setPosition(.25,.5) #Set its position.

mySlider.setTickSize(4,4) #Increase the size.



myBar = viz.addProgressBar('How much?') #Make a bar with its text.

myBar.setPosition(.5,.2) #Set its position.

myBar.set(.5) #Set the current level.
```

Text boxes

Text boxes are fields for users to enter typed data. To make more room for the user to write, use `<text box>.length(<scale>)` or set the overflow properties using `<text`

`box>.overflow(<overflow flag>)`. To set the message inside a text box, use `<text box>.message( <string> )`.

```
#Add a text box.

box = viz.addTextbox()

#Make it twice as long.

box.length(2)

#Place it in window.

box.setPosition(.5,.8)

#Have it grow when text reaches its boundary.

box.overflow(viz.OVERFLOW_GROW)
```

Droplists

Drop lists are drop-down menus. You can add and clear items from a droplist.

```
drop = viz.addDropList() #Create a drop-down list.

drop.setPosition( .5,.5) #Move it.

items = ['Duck', 'Pig', 'Cat', 'Emu'] #Add a list of items.

drop.addItems( items )
```

See also

Getting data from GUI elements

To pull user input from buttons, boxes and sliders, you can call them directly or set up a callback that gets triggered by new input.

Getting data from a buttons, checkboxes and sliders

Use <`GUI element>.get()` to get the data from a button, check box or slider.

- If <`GUI element>` is a **button** or a **checkbox**, the command will return a '0' if the button is down and '1' if it's up.
- If <`GUI element>` is a **slider**, the command will return the tick's position along the bar.

If you want to know if the user is currently moving the tick on a slider, use `<slider>.dragging()`. If the user is dragging the tick, the command will return a '1'.

```
data = mySlider.get() #Get the tick location.

isDragging = mySlider.dragging() #Is the tick dragging?
```

Getting data from a text boxes

The <`GUI element>.get()` command will give you the typed input from a textbox. You can also get the cursor location in the textbox using `<text box>.getCursorPos()`.

```
myTextBox = viz.addTextbox()

myTextBox.get() #Get the text in the box.

myTextBox.getCursorPos() #Get the position of the cursor.
```

Getting data from droplists

Use <droplist>.getSelection() to get the index of the current selection on the droplist. Note: since this command will only give you the current index, you can use <droplist>.getItems() to find out the whole array of items and then use the index to pick out the value of the selected item.

```
#Find the index of the current selection.

drop.getSelection()

#Find the selection itself.
```

```
drop.getItems()[ drop.getSelection() ]
```

Setting up a GUI callback

The action library has methods of registering callback functions for GUI callback events. The commands `<vizact>.onbuttondown`, `<vizact>.onbuttonup`, `<vizact>.onslider`, `<vizact>.onlist`, and `<vizact>.ontextbox` will handle the following events.

- Button events (`viz.BUTTON_EVENT`) : triggered by checkboxes, buttons, radio buttons, or clicking on a drop-down menu.
- Slider events (`viz.SLIDER_EVENT`) : triggered by sliders and progress bars.
- List events (`viz.LIST_EVENT`) : triggered when someone makes a new choice from a drop-down menu (but not when the current selection is re-selected).
- Textbox events (`viz.TEXTBOX_EVENT`) : triggered when text is entered in a text box.

```
#Add and position a slider and progress bar.

mySlider = viz.addSlider()

myProgressBar = viz.addProgressBar('How much?')

mySlider.setPosition(.5,.5)

myProgressBar.setPosition(.5,.25)



#Create functions to handle slider events



def sliderChanged(pos):

    print('mySlider is at ' + str(pos))



vizact.onslider(mySlider,sliderChanged)
```

```
def progressBarChanged(pos):

    print('myProgressBar is at ' + str(pos))



vizact.onslider(myProgressBar,progressBarChanged)
```

See also

GUI appearance

Adding textures to GUI elements

To set the texture that appears when a button or checkbox is selected or unselected, use `<button or checkbox>.downpicture( <texture> )` or `<button or checkbox>.uppicture(<texture>)`, respectively. You can also apply a texture to the tick or bar on a slider.

```
#Add a slider.

mySlider = viz.addSlider()

mySlider.setPosition(.5,.25)

mySlider.setTickSize(4,3)

#Give the tick a new texture.

mySlider.tickpicture('vizinfo.tif')



#Add a checkbox.

myCheckBox = viz.addCheckbox()

myCheckBox.setPosition(.5,.75)

#Give it textures for when it's selected and unselected.
```

```
myCheckBox.uppicture('play.tif')

myCheckBox.downpicture('pause.tif')
```

Fonts

To change the font of any of your GUI elements, just use `<GUI element>.font(<font filename>`. Vizard will look inside the Windows/Font directory when setting a font. You can browse that directory to see what fonts are available on your system.

```
#Add a progress bar and choose its font.

myProgressBar = viz.addProgressBar('How is it going?')

myProgressBar.setPosition(.5,.5)

myProgressBar.font('Times New Roman')
```

Changing the color themes

To change the colors of text, background, and borders of GUIs, set their themes. This works for certain GUI elements (progress bars and droplists) and GUI containters (`vizmenu`, `vizdlg`, and `vizinfo`). The themeExample.py script (File > Quick Open > type: themeExample) shows how changing various theme properties affects a menu and info panel.

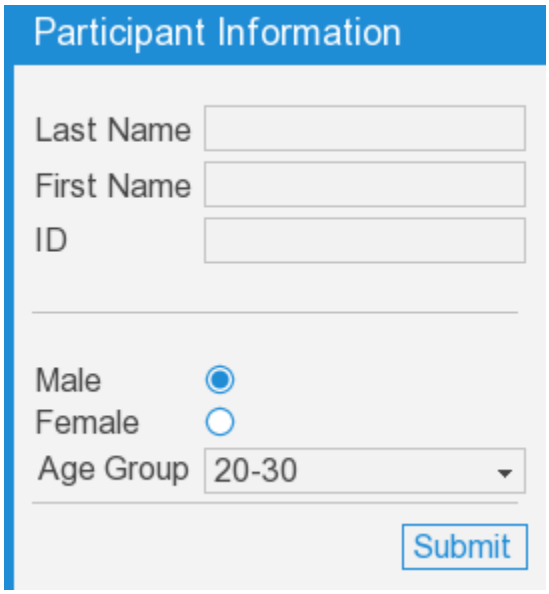
Organizing GUIs with vizinfo

Use the `vizinfo` library to group GUI elements in a panel. It supports lists and custom layouts of elements. Other features include:

- Texture quads and models can be added to the panel.
- If no text is specified when adding the panel, the doc string of the main module is displayed.
- Use a key (default is `F1`) or click the panels icon to toggle the visibility of the panel.

GUI Element List

Use the `<InfoPanel>.addLabelItem()` method to add a GUI element and label that are aligned to the left side of the panel. Use the `<InfoPanel>.addItem()` to add a GUI element without a label and custom alignment. The elements are added to the panel from top to bottom:




```
import viz

import vizinfo



viz.go()



#Add an InfoPanel with a title bar

participantInfo = vizinfo.InfoPanel('',title='Participant
Information',icon=False)



#Add name and ID fields

lastName = participantInfo.addLabelItem('Last Name',viz.addTextbox())
```

```
firstName = participantInfo.addLabelItem('First Name',viz.addTextbox())

ID = participantInfo.addLabelItem('ID',viz.addTextbox())

participantInfo.addSeparator(padding=(20,20))



#Add gender and age fields

male = participantInfo.addLabelItem('Male',viz.addRadioButton(0))

female = participantInfo.addLabelItem('Female',viz.addRadioButton(0))

age = participantInfo.addLabelItem('Age Group',viz.addDropList())

age.addItems(['20-30','31-40','41-50','51-60','61-70'])

participantInfo.addSeparator()



#Add submit button aligned to the right

submitButton =
participantInfo.addItem(viz.addButtonLabel('Submit'),align=viz.ALIGN_RIGHT_CENT
ER)
```

Node3D Objects

In addition to GUI elements, info panels display most `node3D` objects:

```
import viz

import vizinfo



viz.go()



pigeon = viz.addAvatar('pigeon.cfg',scale=[200]*3)
```

```
pigeon.addAction(vizact.spin(0,1,0,45))

pigeon.enable(viz.DEPTH_TEST,op=viz.OP_ROOT)



#Add the pigeon to the panel

panel = vizinfo.InfoPanel('Find the pigeons before they fly
away',align=viz.ALIGN_CENTER,fontSize=22,icon=False)

panel.addItem(pigeon,align=viz.ALIGN_CENTER)
```

Custom GUI Layout

Custom layouts of GUI elements can be created by nesting `vizdlg.Panel` objects within the main `vizinfo` panel. In the following example a `vizdlg` panel with a horizontal layout is used to place three checkboxes side by side:

```
import viz

import vizinfo

import vizdlg



viz.go()



dojo = viz.addChild('dojo.osgb')



info = vizinfo.InfoPanel('An InfoPanel with a sub-panel')



#Add row of checkboxes

row =
vizdlg.Panel(layout=vizdlg.LAYOUT_HORZ_BOTTOM,background=False,border=False)
```

```
check1 = row.addItem(viz.addCheckbox())

check2 = row.addItem(viz.addCheckbox())

check3 = row.addItem(viz.addCheckbox())

info.addLabelItem('Checkboxes',row)
```

Display the Doc String of the Main Module

If no text is specified when adding the info panel, the doc string of the main module is displayed. This is a handy way to both comment the script and display information to the user when it is running:

```
'''

This is a simple example that shows how the doc string

can be displayed in the InfoPanel. Press F1 or click the icon

to toggle the panel.

'''



import viz

import vizinfo

viz.go()



infoPanel = vizinfo.InfoPanel()

dojo = viz.addChild('dojo.osgb')
```

Text and GUI Size

Set the `fontSize` in the info panel constructor to change the main text and title text size. This does not affect the size of text and GUI elements in the body of the panel:

```
import viz

import vizinfo



viz.go()



info = vizinfo.InfoPanel('Main Text',title='Title Text',fontSize=22)

info.addSeparator()



for i in range(1,6):

    info.addLabelItem('Text Box {}'.format(i),viz.addTextbox())
```

Set the `fontSize` of the title bar to change the title text size. This does not affect the size of the main text:

```
info.getTitleBar().fontSize(24)
```

Set the `fontSize` of the underlying panel to change the size of text and GUI elements in the body of the info panel:

```
info.getPanel().fontSize(20)
```

InfoPanel Commands

The following command creates an `InfoPanel` object:

Command	Description

vizinfo.InfoPanel(text=MainDocString, align=viz.ALIGN_LEFT_TOP, margin=(15.0,15.0), key=viz.KEY_F1, icon=True, fontSize=18, window=viz.MainWindow) title=None)	Creates an InfoPanel object. `text`: Sets the text of the info panel. If nothing is specified and the script contains a main docstring (top of the script), the docstring text will be used. `align`: Sets the alignment mode of the info panel. See the table below for alignment options. `margin`: Sets the [x,y] distance of the info panel from the screen edge in pixel units. `key`: Sets the key to toggle the visibility of the info panel. `icon`: Sets the visibility of the icon. `fontSize`: Sets the size of text and GUI elements `window`: Specifies the window that the panel is attached to `title`: Sets the text of the title bar.

The following methods are available on the `InfoPanel` object:

Method	Description

<InfoPanel>.getTextNode()	Gets the underlying text node object.
<InfoPanel>.getPanel()	Gets the underlying `vizdlg.Panel` object.
<InfoPanel>.addItem(item, align=None, visible=True, insert=None, fontSize=None)	`item`: A GUI element, texture quad, or other node3D object. `align`: See the table below for `align` modes. `insert`:Determines the item's order placement on the panel. A value of 0 means the item will be the first one added to the panel. If no value is given the item is added after all other items. `fontSize`:Determines the size of the item. If no value is given the panel `fontSize` value is used.
<InfoPanel>.addSeparator(ratio=0.9,	Draws a horizontal line across the panel. `ratio`: argument defines the line length to panel length ratio. If the value is one, the line is drawn all the way across the panel.

padding=(0,5)	`padding`: defines the spacing in pixels above and below the line.
<InfoPanel>.addLabelItem(name, item)	`name`: text label placed to the left of the item `item`: A GUI element, texture quad, or other node3D object.
<InfoPanel>.setPanelVisible(mode, animate=True)	`mode`: Can be `viz.ON`, `viz.OFF`, or `viz.TOGGLE`
<InfoPanel>.getPanelVisible()	Gets the visibility of the panel.
<InfoPanel>.setAnimation(animation)	Custom animations for toggling the panel's visibility can be created by extending the `vizinfo.InfoAnimation` class.

<InfoPanel>.getAnimation()	Gets the animation for toggling the visibility of the panel
<InfoPanel>.setText(text)	Sets the text of the info panel.
<InfoPanel>.getText()	Gets the text of the info panel.
<InfoPanel>.setTitle(title)	Sets the title text of the info panel. Use None to remove the title bar.
<InfoPanel>.getTitle()	Gets the title text of the info panel, or None if no title bar is assigned.
<InfoPanel>.getTitleBar()	Returns the underlying `vizdlg.TitleBar` object, or None if no title bar is assigned.
<InfoPanel>.setAlignment(align)	Sets the alignment mode of the info panel. See the table below for alignment options.
<InfoPanel>.getAlignment()	Gets the alignment mode of the info panel.

<InfoPanel>.setMargin(margin)	Sets the [x,y] distance of the info panel from the screen edge in pixel units.
<InfoPanel>.getMargin()	Gets the [x,y] distance of the info panel from the screen edge in pixel units.
<InfoPanel>.setToggleKey(key)	Sets the key to toggle the visibility of the info panel.
<InfoPanel>.getToggleKey	Gets the key that toggles the visibility of the info panel.
<InfoPanel>.getIcon()	Gets the icon node object.
<InfoPanel>.setIconTexture(texture)	Sets the texture for the icon.

The following alignment modes are available:

Alignment
viz.ALIGN_LEFT_TOP

viz.ALIGN_RIGHT_TOP
viz.ALIGN_LEFT_BOTTOM
viz.ALIGN_RIGHT_BOTTOM
viz.ALIGN_CENTER
viz.ALIGN_LEFT_CENTER
viz.ALIGN_RIGHT_CENTER
viz.ALIGN_CENTER_TOP
viz.ALIGN_CENTER_BOTTOM

Making pop-downs with vizmenu

To create pop-down menus, use Vizard's vizmenu module and then fill those menus with any of Vizard's GUI elements (sliders, buttons, dropdown lists, text boxes, etc.). You can also

nest pop-down menus within your main pop-downs to create expanding menus. Here's the basic process for creating a pop-down menu:

```
#Create main menu object

import vizmenu

menu = vizmenu.add()

#Align the menu in the enter of the top of the screen.

menu.setAlignment( vizmenu.CENTER )

#Add two pop-downs menus within the main menu.

AppearanceMenu = menu.add( 'Appearance' )

Ball1Menu = menu.add( 'Ball 1' )



#Add a droplist to one of the pop-downs.

themeDropDown = AppearanceMenu.add( viz.DROPLIST, 'Theme' )

themeDropDown.addItems( ['Default','Dark','Green'] )



#Add sub-menu under the Ball1 menu and put radio buttons in it.

Ball1ColorMenu = Ball1Menu.add( vizmenu.MENU, 'Color' )

Ball1Red = Ball1ColorMenu.add( viz.RADIO, 0, 'Red' )

Ball1White = Ball1ColorMenu.add( viz.RADIO, 0, 'White' )

Ball1Blue = Ball1ColorMenu.add( viz.RADIO, 0, 'Blue' )

#Add another sub-menu to the Ball1 menu for a slider.

Ball1SizeSlider = Ball1Menu.add( viz.SLIDER, 'Size' )
```

To change the appearance of your pop-downs (the color theme, fonts, etc.), check out the section on GUI appearance. To learn how to get the user responses from the elements in the pop-down menu, check out the section on getting data from GUI elements.

vizconfig basics

Use the `vizconfig` module to group together objects in the virtual world and their related GUI controls. A tabbed window displays GUI elements which can include text boxes, buttons, check boxes, progress bars, drop lists, and radio buttons.

Under each tab in a configuration window appears the User Interface for a configurable object. A configurable object contains related GUI elements, their layout and information about what to do when those GUIs are updated. In this section objects of the `BasicConfigurable` class will be covered.

Once an object of the `BasicConfigurable` class is created a variety of methods to add GUI items can be called. These methods do more than just add the basic GUI elements. Many of them provide additional functionality (e.g. text boxes that accept either float,int, or string values). Event handling is built in, so it is only necessary to specify the function that gets called when the GUI is updated. A function whose return value will set the GUI state can also be specified when adding the item. The GUI elements are added in a list format with a name appearing on the left and the element on the right.

This UI of the `BasicConfigurable` object will be displayed and active when the object is added to a configuration window. There are methods available on this window object for adding/removing configurables, setting it's position on the screen, and controlling it's visibility.

> **Note**: By default you must press F12 to toggle the visibility of the configuration window.

The following example code shows how to add a `BasicConfigurable` object with several GUI items:

```
'''
```

```
Press F12 to toggle the visibility of the configuration window

'''


import viz

import vizconfig

import vizshape

import vizinfo


viz.setMultiSample(4)

viz.fov(60)

viz.go()


vizinfo.InfoPanel()


gallery = viz.add('gallery.osgb')


#add an object that the user can manipulate with the UI

ball = vizshape.addSphere(radius=0.1,pos=[0,1.8,2])


#Create a configurable object for the ball

bc= vizconfig.BasicConfigurable('ball')

bc.addBoolItem('visible',ball.visible,ball.getVisible)

bc.addFloatRangeItem('alpha',[0,1],fset=ball.alpha,fget=ball.getAlpha)
```

```
bc.addChoiceRadioItem('color',[('white',[1,1,1]),('red',[1,0,0]),('blue',[0,0,1
])],ball.color,ball.getColor)



#register the object so it will display in the configuration window

vizconfig.register(bc)
```

The following command creates an object of the `BasicConfigurable` class:

Command	Description
vizconfig.BasicConfigurable(name)	Returns an object of the BasicConfigurable class. Once this object has been registered with a configuration window, the string passed in the name argument will be displayed on a tab in the window.

The following methods add a GUI element to a `BasicConfigurable` object:

Method	Description
<BasicConfigurable>.addInteractiveItem(name, object)	Creates an interactive shell that allows the user to execute Python commands on the object specified.

<BasicConfigurable>.addCommandItem(name, label, fdown=None, fup=None)	Creates a button that will call the specified functions when pressed/released. The label value is a string that labels the button.
<BasicConfigurable>.addBoolItem(name, fset=None, fget=None)	Creates a checkbox. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the checkbox state.
<BasicConfigurable>.addIntItem(name, range=[None,None],	Creates a textbox that will allow the user to enter integer values. If a range is given only values within that range will be accepted. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.

fset=None, fget=None)	
<BasicConfigurable>.addFloatItem(name, range=[None,None], fset=None, fget=None)	Creates a textbox that will allow the user to enter float values. If a range is given only values within that range will be accepted. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.
<BasicConfigurable>.addIntRangeItem(name, range, fset=None, fget=None)	Creates a progress bar that will allow the user to control an integer range. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the progress bar position.

<BasicConfigurable>.addFloatRangeItem(name, range, fset=None, fget=None)	Creates a progress bar that will allow the user to control an integer range. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the progress bar position.
<BasicConfigurable>.addTextItem(name, fset=None, fget=None)	Creates a textbox. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.
<BasicConfigurable>.addChoiceListItem(name. choices,	Creates a droplist that will allow the user to select from a list of choices. The choices argument accepts a two dimensional list. The first position of each element is the name that appears in the droplist. The second position is it's value. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function

fset=None, fget=None)	specified in the fget argument will set the droplist selection.
<BasicConfigurable>.addChoiceRadioItem(name, choices, fset=None, fget=None)	Creates a group of radio buttons that will allow the user to select one radio button at a time. The choices argument accepts a two dimensional list. The first position of each element is the label that appears next to the radio button. The second position is it's value. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will determine which radio button is active.

The following methods are also available on a `BasicConfigurable` object:

Method	Description
<BasicConfigurable>.updateConfigUI()	Updates the UI state of all items belonging to the configurable. This will call the function specified in the fget argument of each item.

<BasicConfigurable>.updateConfigItem(name)	Updates the UI state of a specific GUI item belonging to the configurable. This will call the function specified in the fget argument of the item.
<BasicConfigurable>.getConfigName()	Returns the name of the configurable.

The following commands return a configuration window and register/unregister configurable objects with a configuration window:

Command	Description
vizconfig.getConfigWindow(name=DEFAULT_WINDOW)	Returns the config window with the specified name. If a config window with that name does not exist, one will be created.
vizconfig.register(obj, window=DEFAULT_WINDOW, name=None)	Adds a configurable object to a config window. If a config window with that name does not exist, one will be created. The display name of the configurable can be optionally overridden.
vizconfig.unregister(obj, window=DEFAULT_WINDOW)	Removes a configurable object from a config window.

The following methods are available on the `ConfigWindow` object:

Method	Description
<ConfigWindow>.addConfigurable(config, name=None)	Adds a configurable object to the config window. The display name of the configurable can be optionally overridden.
<ConfigWindow>.removeConfigurable(config)	Removes a configurable object from the config window.
<ConfigWindow>.setWindowVisible(visible)	Sets the visibility of the config window. The following values are accepted: True, False, or viz.TOGGLE. By default the visibility of the config window is set to False.
<ConfigWindow>.getWindowVisible()	Gets the visibility of the config window. Either True or False will be returned.
<ConfigWindow>.setWindowAlignment(align)	Sets the alignment mode of the config window. See the table below for possible align values. The default alignment mode is vizconfig.ALIGN_RIGHT_TOP

<ConfigWindow>.getWindowAlignment()	Returns the alignment mode of the config window.
<ConfigWindow>.setWindowMargin(pixels)	Sets the [x,y] distance of the config window from the screen edge in pixel units. The default value is [10.0,10.0].
<ConfigWindow>.getWindowMargin()	Gets the [x,y] distance of the config window from the screen edge in pixel units.
<ConfigWindow>.setWindow(window)	Sets the window to draw the config window into.
<ConfigWindow>.getWindow()	Gets the window that the config window draws into
<ConfigWindow>.setActivationKey(key)	Sets the key that will toggle the visibility of the config window. The default value for this key is F12.
<ConfigWindow>.getActivationKey()	Gets the key that will toggle the visibility of the config window.

Use one of the following alignment modes for the align argument when calling the `<ConfigWindow>.setWindowAlignment()` method:

Mode
vizconfig.ALIGN_LEFT_TOP
vizconfig.ALIGN_RIGHT_TOP
vizconfig.ALIGN_LEFT_BOTTOM
vizconfig.ALIGN_RIGHT_BOTTOM
vizconfig.ALIGN_CENTER
vizconfig.ALIGN_LEFT_CENTER
vizconfig.ALIGN_RIGHT_CENTER
vizconfig.ALIGN_CENTER_TOP

```
vizconfig.ALIGN_CENTER_BOTTOM
```

vizconfig advanced

To create configurable objects that encapsulate both `vizconfig` GUI items and objects in the virtual world write your own class that extends from `Configurable`.

Note: In order to follow this section some knowledge of object oriented programming is required.

Extending the Configurable class

The `BasicConfigurable` class discussed in the last section provides a wrapper for `Configurable` objects and implements a table like UI for displaying GUI elements. Extending the base class `Configurable` directly and implementing the same table UI is not difficult to do. Objects in the virtual world can be added as data attributes to the derived class and other customizations can be made.

This is demonstrated in the following code. A ball model and methods related to it's rotation speed are added to the `ballObject` configurable. The table UI is created in the `CreateConfigGUI` method, which is automatically called when the `Configurable` is registered:

```
import viz
import vizconfig


class ballObject(vizconfig.Configurable):
    def __init__(self):
        vizconfig.Configurable.__init__(self,'Ball')
```

```
        self.ball = viz.add('ball.wrl',pos=(0,1.5,2))
        self.setSpeed(10)

        #Automatically register configurable
        vizconfig.register(self)

    def setSpeed(self,val):
        self.ball.speed = val
        self.ball.runAction(vizact.spin(0,1,0,val))
        self.updateConfigItem('speed')

    def getSpeed(self):
        return self.ball.speed

    def createConfigUI(self):
        ui = vizconfig.DefaultUI()
        ui.addBoolItem('visible',self.ball.visible,self.ball.getVisible)
ui.addFloatRangeItem('speed',[-180.0,180.0],self.setSpeed,self.getSpeed)
ui.addConfigItem('texture',viz.addTexQuad(texture=viz.add('ball.jpg'),size=30))
        return ui
```

```
viz.go()



gallery = viz.add('gallery.ive')

ball = ballObject()



#Manually set speed which will automatically update UI

vizact.onkeydown('s',ball.setSpeed,0)
```

The virtual interface of the `Configurable` class contains the following methods that can be overridden:

Method	Description
<Configurable>.createConfigUI()	Use this method to create an object that contains UI controls for this configurable.
<Configurable>.onConfigActivate()	This will be called when the configurable UI is activated.
<Configurable>.onConfigDeactivate()	This will be called when the configurable UI is deactivated

`Configurable` objects also contains the following methods:

Method	Description

<Configurable>.updateConfigUI()	Updates the UI state of all items belonging to the configurable.
<Configurable>.updateConfigItem(name)	Updates the UI state of a specific item belonging to the configurable.
<Configurable>.getConfigName()	Gets the name of the configurable.
<Configurable>.getConfigUI()	Returns the root UI node for the configurable. Can be None if not yet created.
<Configurable>.getOrCreateConfigUI()	Returns the root UI node for the configurable. If one does not exist the createConfigUI() method will be called.

Creating the User Interface for a Configurable object

The UI of a `Configurable` object is defined when it's `CreateConfigGUI` method is called. This occurs automatically when the object is registered with a configuration window. Any two dimensional Vizard node that is returned from this method will display in the configuration window.

The `vizconfig` module provides a `DefaultUI` object that manages items and displays the elements in a table format with names on the left and the item on the right. To create a `DefaultUI` object use the following:

Command	Description
vizconfig.DefaultUI()	Returns a table UI that manages and displays GUI items and other node3D objects.

A `DefaultUI` object has the following methods available:

Method	Description
<DefaultUI>.addConfigItem(name, item)	Adds any node3D object to the UI (e.g. display an image in the UI by creating a texture quad).
<DefaultUI>.addInteractiveItem(name, object)	Creates an interactive shell that allows the user to execute Python commands on the object specified.
<DefaultUI>.addCommandItem(name, label,	Creates a button that will call the specified functions when pressed/released. The

fdown=None, fup=None)	label value is a string that labels the button.
<DefaultUI>.addBoolItem(name, fset=None, fget=None)	Creates a checkbox. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the checkbox state.
<DefaultUI>.addIntItem(name, range=[None,None], fset=None, fget=None)	Creates a textbox that will allow the user to enter integer values. If a range is given only values within that range will be accepted. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.
<DefaultUI>.addFloatItem(name, range=[None,None], fset=None, fget=None)	Creates a textbox that will allow the user to enter float values. If a range is given only values within that range will be accepted. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.

<DefaultUI>.addIntRangeItem(name, range, fset=None, fget=None)	Creates a progress bar that will allow the user to control an integer range. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the progress bar position.
<DefaultUI>.addFloatRangeItem(name, range, fset=None, fget=None)	Creates a progress bar that will allow the user to control an integer range. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the progress bar position.
<DefaultUI>.addTextItem(name, fset=None, fget=None)	Creates a textbox. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will appear in the textbox.
<DefaultUI>.addChoiceListItem(name. choices,	Creates a droplist that will allow the user to select from a list of choices. The choices argument accepts a two dimensional list. The first position of each element is the name that appears in the droplist. The

	fset=None, fget=None)	second position is it's value. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will set the droplist selection.
<DefaultUI>.addChoiceRadioItem(name,	choices, fset=None, fget=None)	Creates a group of radio buttons that will allow the user to select one radio button at a time. The choices argument accepts a two dimensional list. The first position of each element is the label that appears next to the radio button. The second position is it's value. The fset argument specifies the function to call when the GUI has been updated. The value returned by the function specified in the fget argument will determine which radio button is active.

Note: If you wish to change the layout of items, this is possible by extending from the base class `ConfigUI` and arranging the items in the derived class. Take a look at the `TableUI` class in `vizconfig.py` for an example of this.

vizdlg introduction

Use the `vizdlg` library to add panels with GUIs and dialog objects to the screen:

- `Panel`: Draws a box that can arrange and display any `node3D` object including other `Panels`, GUI elements, dialog components, or texture quads. The appearance of the `Panel` and GUI elements within can be changed by setting color themes and font sizes.

- `TabPanel`: Creates a `Panel` with tabs.
- `GridPanel`: Creates a `Panel` that arranges components in a table format.

Dialog objects are panels with accept/cancel buttons:

- `InputDialog`: Adds a prompt and textbox for input.
- `TickerDialog`: Allows the user to select from a range of numbers with up and down buttons.
- `ColorDialog`: Allows the user to select RGB values and displays the resulting color.
- `TextureDialog`: Allows the user to select from a group of textures.

Example

The following example code shows how to create a simple `Panel` object with several GUI elements:

```
import viz

import vizdlg



viz.go()



myPanel = vizdlg.Panel()



myPanel.addItem(viz.addText('A simple panel'),align=vizdlg.ALIGN_CENTER)

myPanel.addItem(viz.addCheckbox())

myPanel.addItem(viz.addSlider())

myPanel.addItem(viz.addTextbox())
```

```
viz.link(viz.CenterCenter,myPanel,offset=(-100,50,0))
```

The image below shows a `TabPanel` with a `TickerDialog` in the active tab:



The following code creates the panel above:

```
import viz

import vizdlg



viz.go()



p1 =
vizdlg.TickerDialog(label='Circle',units='pixels',range=(1,5,1),editable=True,b
order=False,background=False,margin=0)

p2 =
vizdlg.TickerDialog(label='Crosshair',units='pixels',range=(1,5,1),editable=Tru
e,border=False,background=False,margin=0)

p3 =
vizdlg.TickerDialog(label='Border',units='pixels',range=(1,5,1),editable=True,b
order=False,background=False,margin=0)



tp = vizdlg.TabPanel(align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('Circle',p1,align=vizdlg.ALIGN_LEFT_TOP)

tp.addPanel('Crosshair',p2,align=vizdlg.ALIGN_CENTER_TOP)
```

```
tp.addPanel('Border.',p3,align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('checkbox',viz.addCheckbox(),align=vizdlg.ALIGN_LEFT_BOTTOM)

tp.addPanel('slider',viz.addSlider(),align=vizdlg.ALIGN_CENTER_BOTTOM)

tp.addPanel('movie',viz.addTexQuad(size=75,texture=viz.add('vizard.mpg',loop=True,play=True)),align=vizdlg.ALIGN_RIGHT_BOTTOM)



viz.link(viz.RightTop,tp,offset=(-20,-20,0))
```

vizdlg introduction

Use the `vizdlg` library to add panels with GUIs and dialog objects to the screen:

- **`Panel`: Draws a box that can arrange and display any `node3D` object including other `Panels`, GUI elements, dialog components, or texture quads. The appearance of the `Panel` and GUI elements within can be changed by setting color themes and font sizes.**
- **`TabPanel`: Creates a `Panel` with tabs.**
- **`GridPanel`: Creates a `Panel` that arranges components in a table format.**

Dialog objects are panels with accept/cancel buttons:

- **`InputDialog`: Adds a prompt and textbox for input.**
- **`TickerDialog`: Allows the user to select from a range of numbers with up and down buttons.**
- **`ColorDialog`: Allows the user to select RGB values and displays the resulting color.**
- **`TextureDialog`: Allows the user to select from a group of textures.**

Example

The following example code shows how to create a simple `Panel` object with several GUI elements:

```
import viz

import vizdlg



viz.go()



myPanel = vizdlg.Panel()



myPanel.addItem(viz.addText('A simple panel'),align=vizdlg.ALIGN_CENTER)

myPanel.addItem(viz.addCheckbox())

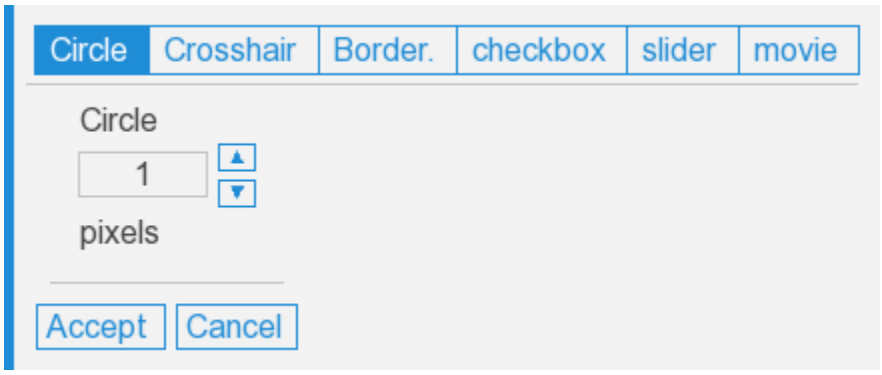
myPanel.addItem(viz.addSlider())

myPanel.addItem(viz.addTextbox())



viz.link(viz.CenterCenter,myPanel,offset=(-100,50,0))
```

The image below shows a `TabPanel` with a `TickerDialog` in the active tab:



The following code creates the panel above:

```
import viz
```

```
import vizdlg



viz.go()



p1 =
vizdlg.TickerDialog(label='Circle',units='pixels',range=(1,5,1),editable=True,
border=False,background=False,margin=0)

p2 =
vizdlg.TickerDialog(label='Crosshair',units='pixels',range=(1,5,1),editable=Tr
ue,border=False,background=False,margin=0)

p3 =
vizdlg.TickerDialog(label='Border',units='pixels',range=(1,5,1),editable=True,
border=False,background=False,margin=0)



tp = vizdlg.TabPanel(align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('Circle',p1,align=vizdlg.ALIGN_LEFT_TOP)

tp.addPanel('Crosshair',p2,align=vizdlg.ALIGN_CENTER_TOP)

tp.addPanel('Border.',p3,align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('checkbox',viz.addCheckbox(),align=vizdlg.ALIGN_LEFT_BOTTOM)

tp.addPanel('slider',viz.addSlider(),align=vizdlg.ALIGN_CENTER_BOTTOM)

tp.addPanel('movie',viz.addTexQuad(size=75,texture=viz.add('vizard.mpg',loop=T
rue,play=True)),align=vizdlg.ALIGN_RIGHT_BOTTOM)



viz.link(viz.RightTop,tp,offset=(-20,-20,0))
```

vizdlg introduction

Use the `vizdlg` library to add panels with GUIs and dialog objects to the screen:

- `Panel`: Draws a box that can arrange and display any `node3D` object including other `Panels`, GUI elements, dialog components, or texture quads. The appearance of the `Panel` and GUI elements within can be changed by setting color themes and font sizes.
- `TabPanel`: Creates a `Panel` with tabs.
- `GridPanel`: Creates a `Panel` that arranges components in a table format.

Dialog objects are panels with accept/cancel buttons:

- `InputDialog`: Adds a prompt and textbox for input.
- `TickerDialog`: Allows the user to select from a range of numbers with up and down buttons.
- `ColorDialog`: Allows the user to select RGB values and displays the resulting color.
- `TextureDialog`: Allows the user to select from a group of textures.

Example

The following example code shows how to create a simple `Panel` object with several GUI elements:

```
import viz

import vizdlg



viz.go()



myPanel = vizdlg.Panel()



myPanel.addItem(viz.addText('A simple panel'),align=vizdlg.ALIGN_CENTER)

myPanel.addItem(viz.addCheckbox())
```

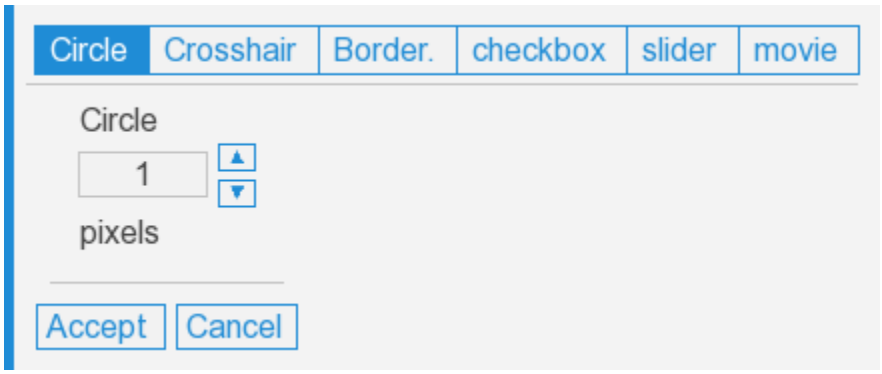
```
myPanel.addItem(viz.addSlider())

myPanel.addItem(viz.addTextbox())



viz.link(viz.CenterCenter,myPanel,offset=(-100,50,0))
```

The image below shows a `TabPanel` with a `TickerDialog` in the active tab:



The following code creates the panel above:

```
import viz

import vizdlg



viz.go()



p1 =
vizdlg.TickerDialog(label='Circle',units='pixels',range=(1,5,1),editable=True,b
order=False,background=False,margin=0)

p2 =
vizdlg.TickerDialog(label='Crosshair',units='pixels',range=(1,5,1),editable=Tru
e,border=False,background=False,margin=0)

p3 =
vizdlg.TickerDialog(label='Border',units='pixels',range=(1,5,1),editable=True,b
order=False,background=False,margin=0)
```

```
tp = vizdlg.TabPanel(align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('Circle',p1,align=vizdlg.ALIGN_LEFT_TOP)

tp.addPanel('Crosshair',p2,align=vizdlg.ALIGN_CENTER_TOP)

tp.addPanel('Border.',p3,align=vizdlg.ALIGN_RIGHT_TOP)

tp.addPanel('checkbox',viz.addCheckbox(),align=vizdlg.ALIGN_LEFT_BOTTOM)

tp.addPanel('slider',viz.addSlider(),align=vizdlg.ALIGN_CENTER_BOTTOM)

tp.addPanel('movie',viz.addTexQuad(size=75,texture=viz.add('vizard.mpg',loop=True,play=True)),align=vizdlg.ALIGN_RIGHT_BOTTOM)



viz.link(viz.RightTop,tp,offset=(-20,-20,0))
```

vizdlg advanced

By extending the `vizdlg.Dialog` class, you can add GUIs and other components to a `Panel` and use the buttons provided to create a custom form.

> In order to follow this section some knowledge of object oriented programming is required.

All dialog components discussed in the last section extend `vizdlg.Dialog`. A `vizdlg.Dialog` object creates the following: A panel that content can be added to using the `<Dialog>.content` attribute, `accept/cancel` buttons, a `show` method that displays the panel and waits for a button press, and a `remove` method that removes the object.

The following example script shows how to extend `vizdlg.Dialog` and add content. One of the buttons is renamed and the other is hidden. The `viztask` function `yields` until the `show` method is complete and the form data is retrieved and printed.

```
import viz

viz.go()
```

```
import vizdlg



class MyForm(vizdlg.Dialog):



    def __init__(self,**kw):



        #Initialize base class

        vizdlg.Dialog.__init__(self,**kw)



        #Add row of checkboxes

        row =
vizdlg.Panel(layout=vizdlg.LAYOUT_HORZ_BOTTOM,border=False,background=False,mar
gin=0)

        row.addItem(viz.addText('Checkboxes'))

        self.check1 = row.addItem(viz.addCheckbox())

        self.check2 = row.addItem(viz.addCheckbox())

        self.check3 = row.addItem(viz.addCheckbox())



        #Add row to dialog content section

        self.content.addItem(row)



        #Add a subgroup containing slider/textbox

        group = vizdlg.Panel()
```

```
        #Add row for slider to subgroup

        row =
vizdlg.Panel(layout=vizdlg.LAYOUT_HORZ_BOTTOM,border=False,background=False,mar
gin=0)

        row.addItem(viz.addText('Slider'))

        self.slider = row.addItem(viz.addSlider())

        group.addItem(row)



        #Add row for textbox to subgroup

        row =
vizdlg.Panel(layout=vizdlg.LAYOUT_HORZ_BOTTOM,border=False,background=False,mar
gin=0)

        row.addItem(viz.addText('Textbox'))

        self.textbox = row.addItem(viz.addTextbox())

        group.addItem(row)



        #Add group to dialog content section

        self.content.addItem(group)



        #Hide cancel button

        self.cancel.visible(0)



        #Rename accept button to 'submit'

        self.accept.message('Submit')
```

```
import viztask

import vizinfo



def FormTask():



    info = vizinfo.InfoPanel("Press the 's' key to show the form")



    #Create input form

    form = MyForm(title='Form title')



    #Link form to center of screen

    viz.link(viz.MainWindow.CenterCenter,form)



    while True:



        #Wait for 's' key to be pressed

        yield viztask.waitKeyDown('s')



        #TODO: Pause simulation



        #Display form

        yield form.show()
```

```
        if form.accepted:
            #User pressed 'Submit', process data
print('Checkboxes:',form.check1.get(),form.check2.get(),form.check3.get())
            print('Slider:',form.slider.get())
            print('Textbox:',form.textbox.get())

        #TODO: Resume simulation

    #Remove form when completely finished with it
    form.remove()

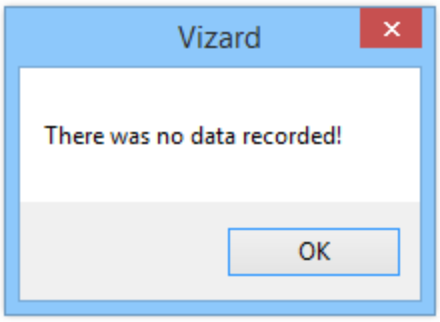
viztask.schedule( FormTask() )
```

vizinput dialogs

The `vizinput` module provides several dialog style boxes for eliciting user input and displaying messages. All dialogs create a blocking action that halts events until the user has responded or the box is closed. To set their positions and titles use the standard dialog options described in the vizinput properties section.

Display a message

Use `vizinput.message()` to display a message box to the user:

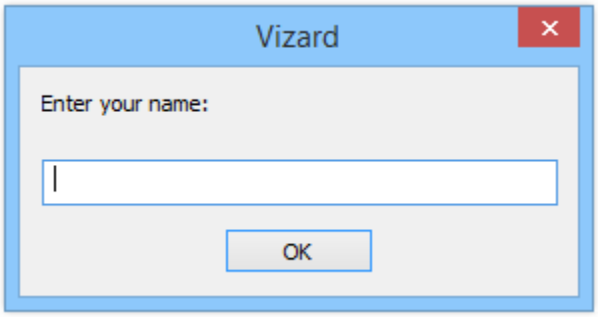



```
import vizinput

vizinput.message('There was no data recorded!')
```

Prompt for a string

The `vizinput.input()` dialog returns a string entered by the user:



The `vizinput.input()` command accepts the following optional keyword arguments, along with the standard dialog options.

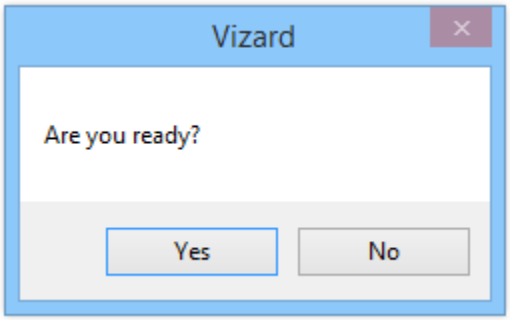
Argument	Description
password	If the value of this argument evaluates to True, then the contents of input control will be displayed as asterisks (*). This is useful when entering passwords or other sensitive information.

```
import vizinput

userName = vizinput.input('Enter your name:')
```

Ask the user a Yes/No question

The `vizinput.ask()` dialog displays a question and allows the user to select a Yes/No response. The answer will be 1 for 'Yes' and 0 for 'No':

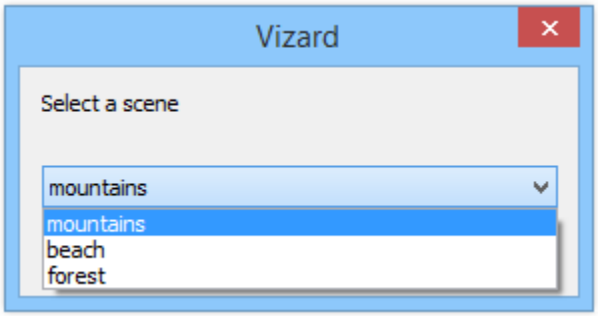



```
import vizinput

answer = vizinput.ask('Are you ready?')
```

Prompt the user to choose from a list of options

Use the `vizinput.choose()` dialog to create a pop-down menu of choices. The user's answer will be the order of their response within the array of options (0 is returned for choice 1, 1 for choice two, etc...):

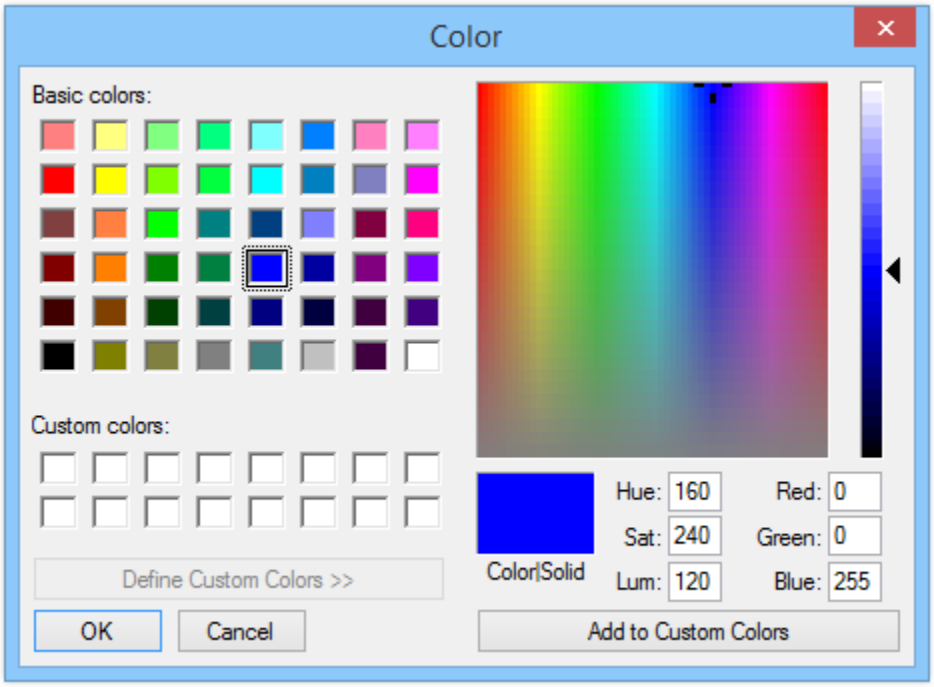



```
import vizinput

choice = vizinput.choose('Select a scene',['mountains','beach','forest'])
```

Prompt the user to select a color

The `vizinput.color()` dialog allows the user to select a color using a variety of tools. The return value is a list of normalized RGB values. Optionally, one can specify the initial color that is selected when the dialog opens.



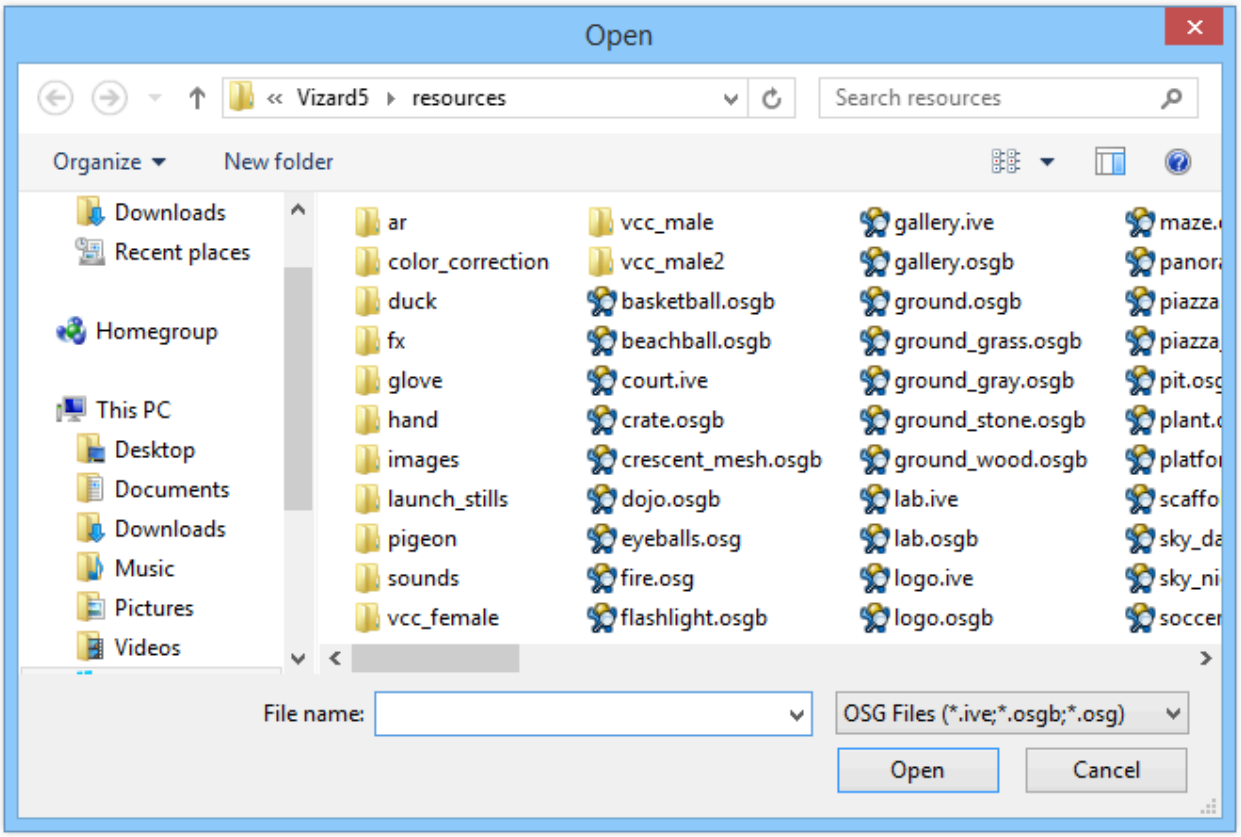

```
import vizinput

color = vizinput.color(viz.BLUE) # Specifying initial color is optional

viz.clearcolor(color)
```

Windows open/save dialogs

Use `vizinput.fileOpen()` and `vizinput.fileSave()` to display standard Windows open and save dialogs. A string holding the path to the selected file is returned when the open or save button is pressed:



The following commands create open/save dialogs:

Command	Description
vizinput.fileOpen()	Displays a Windows open dialog box. The path to the file is returned as a string when the Open button is pressed.
vizinput.fileSave()	Displays a Windows save dialog box. The path to the file is returned as a string when the Save button is pressed.

Both commands accept the following optional keyword arguments, along with the standard dialog options.

Argument	Description
filter	A list of 2 item tuples. The first item of the tuple is a file type description. The second item of the tuple is a file type filter, with wildcards supported.
file	Specifies the default file name to open/save.
directory	Specifies an initial directory to display in the dialog.

In the code below a `vizinput.fileOpen` dialog displays the Vizard resource directory and filters for OSG and VRML files. The file selected is added to a Vizard scene:

```
import viz

import vizinput



filePath = vizinput.fileOpen(filter=[('OSG Files','*.ive;*.osgb;*.osg'),('VRML Files','*.wrl')],directory='c:/Program Files/WorldViz/Vizard4/resources')

model = viz.add(filePath)



viz.go()
```

In the following code a default file name is specified:

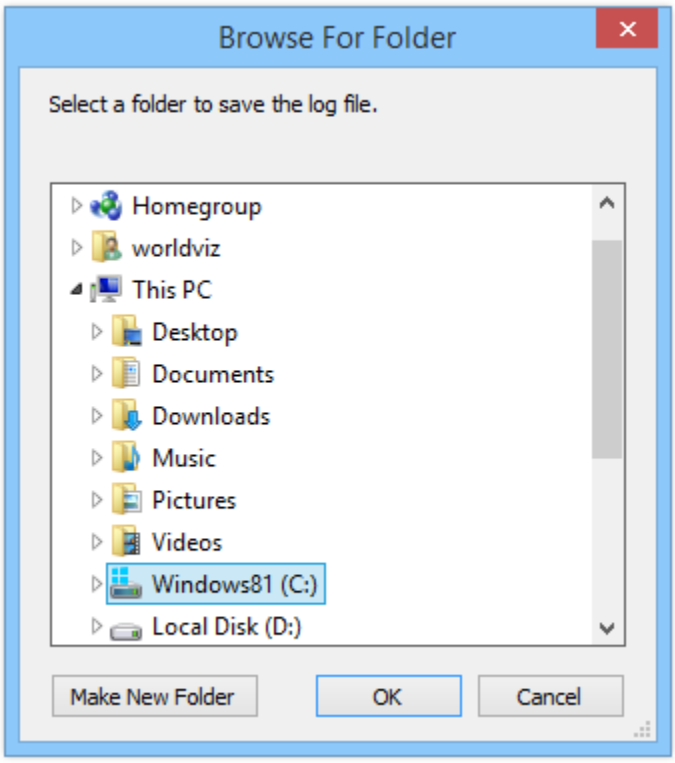
```
import vizinput

filePath = vizinput.fileSave(file='test.py')
```

```
print(filePath)
```

Windows directory selection dialog

The `vizinput.directory()` command displays the standard Windows directory selection dialog. A string holding the path to the selected directory is returned when the OK button is pressed:



The `vizinput.directory()` command accepts the following optional keyword arguments, along with the standard dialog options.

Command	Description
prompt	The message prompt to display above the directory tree.

directory	The initial directory to have selected.

In the code below, a file is opened in the selected directory:

```
import vizinput

path = vizinput.directory(prompt='Select a folder to save the log
file.',directory='C:\\')



file = open(path + '/log1.txt', 'w')
```

Changing the title, parent, and position of a dialog

All `vizinput` dialog objects accept the same optional keyword arguments for changing their title, parent object, and position. The following table lists these arguments and their default values:

Argument	Description
title = 'Vizard'	A string value that is displayed in the upper left corner of the dialog.
parent = viz.window.getHandle()	This can either be the Vizard graphics window or the desktop. By default it is set to the Vizard graphics window. When parenting to the desktop set this argument to 0

center = vizinput.CENTER	Sets the dialog's position in relation to the parent object. See the table below for a list of available flags.
pos = None	This will set the x and y position of the dialog in pixel coordinates, where (0,0) is the upper left corner of the monitor. If a value is given here, the center argument will be ignored.

The following flags and combinations of flags can be used with the `center` argument:

Flag
vizinput.CENTER
vizinput.CENTER_LEFT
vizinput.CENTER_RIGHT
vizinput.CENTER_TOP
vizinput.CENTER_BOTTOM

vizinput.CENTER_LEFT|vizinput.CENTER_TOP
vizinput.CENTER_RIGHT|vizinput.CENTER_TOP
vizinput.CENTER_LEFT|vizinput.CENTER_BOTTOM
vizinput.CENTER_RIGHT|vizinput.CENTER_BOTTOM

Example

```
import viz

viz.go()



import vizinput

vizinput.message('Parented to the Vizard Window, top
center',center=vizinput.CENTER_TOP)

vizinput.message('Parented to the Desktop, top
center',parent=0,center=vizinput.CENTER_TOP)

vizinput.message('Positioned using pixel coordinates',pos=[50,50])

vizinput.message('A different title', title='myApp')
```

Changing default properties

The `vizinput` module provides commands to change the `title`, `parent`, and `center` defaults. This is useful when you need to make a change to multiple dialogs. Instead of using a keyword argument each time one is created, the default command is called once and is applied to all dialogs that follow:

Command	Description
vizinput.setDefaultTitle(title)	**title:** A string that sets the default title.
vizinput.setDefaultParent(parent)	**parent:** Sets the default parent object. Use 0 to parent to the desktop and **viz.window.getHandle()** to parent to the Vizard graphics window
vizinput.setDefaultCenter(center)	**center:** Sets the default center.

Example

The following example sets new `center` and `title` defaults:

```
import viz

viz.go()



import vizinput



#Set new default center
```

```
vizinput.setDefaultCenter(vizinput.CENTER_BOTTOM)



vizinput.message('New default center')

vizinput.color(viz.RED)



#Set new default title

vizinput.setDefaultTitle('MyApp')



vizinput.message('New default title')

vizinput.choose('select...',['1','2','3'])
```

Displaying HTML

One handy way to display instructions to users is by displaying an HTML page on top of the application window. To do so, use the viz.window.displayhtml(<html page>) command. To make the page disappear, use viz.window.hidehtml(<html page>).

```
#Set the application window to fullscreen.

viz.go(viz.FULLSCREEN)

#Display web page.

viz.window.displayHTML( 'www.worldviz.com' )

#Add a keystroke to hide the HTML.

vizact.onkeydown(' ', viz.window.hideHTML )
```

To create a help file for a given script, include an html file with your script's name and ending in the extension '.html' (e.g. 'myscript.html') in your script's directory. When users hit the F1 key while your world is running, the html file will be displayed.

vizhtml introduction

The `vizhtml` module provides three types of functionality:

- Displaying an HTML form in an embedded HTML window and collecting data from it.
- Collecting data from HTML forms submitted from remote browsers on the local network.
- Real time bidirectional communication with remote browsers using WebSockets.

HTML forms are commonly used to pass data from a browser to a web server. Data is entered through input fields (e.g. text fields, check boxes, radio buttons) and submitted with a button click. The `vizhtml` module supports collecting data from HTML forms displayed in the Vizard graphics window and within a remote browser on the local network.

WebSockets are a new technology, so you will need a browser that supports it. Using `vizhtml's` support for WebSocket communication, data can be sent back and forth between Vizard and a browser. For example, someone could get information about a running Vizard program through a web page and simultaneously send feedback or instructions to the user.

HTML forms in Vizard

To display HTML forms in Vizard an HTML window is embedded in the graphics window. The HTML can be specified by filename, URL, or a string variable that holds HTML code. A form submission event will get triggered once a form is submitted and data can be collected using a task or callback function. Once a form is submitted, either the HTML window will be hidden or an HTML reply page can be displayed.

Commands

The following commands are related to displaying HTML in the embedded window:

Command	Description

vizhtml.displayURL(filename, pos = None size = None)	Displays an HTML file in an embedded HTML window. This will trigger a **vizhtml.FORM_SUBMIT_EVENT** when a form submission occurs on the page. **filename:** The name of the HTML file or the URL to display in the HTML window. **pos**: If specified, this sets the position of the HTML window. The position corresponds to the upper left corner of the HTML window relative to the upper left corner of the graphics window. If not specified, the HTML window will be centered in the graphics window. **size:** If specified, this sets the size of the HTML window. If not specified, the size will be set to 2/3 the size of the graphics window.

vizhtml.displayCode(code, pos = None size = None)	Displays HTML code in an embedded HTML window. This will trigger a **vizhtml.FORM_SUBMIT_EVENT** when a form submission occurs on the page. **code:** A string that contains HTML code to display in the HTML window. **pos**: If specified, this sets the position of the HTML window. The position corresponds to the upper left corner of the HTML window relative to the upper left corner of the graphics window. If not specified, the HTML window will be centered in the graphics window. **size:** If specified, this sets the size of the HTML window. If not specified, the size will be set to 2/3 the size of the graphics window.
vizhtml.setFormReplyURL(filename)	Sets the HTML file that will be displayed after a form submission. If None, the embedded HTML window is hidden after the form submission.

	filename: The name of the HTML file or the URL to display.
vizhtml.getFormReplyURL()	Gets the HTML filename that will be displayed after a form submission.
vizhtml.setFormReplyCode(code)	Sets the HTML code that will be displayed after a form submission. If None, the embedded window is hidden after the form submission. **code:** A string that contains HTML code.
vizhtml.getFormReplyCode()	Gets the HTML code that will be displayed after a form submission.

Form Definition

To define forms, you must use the HTML `<form>` tag. All input elements defined within the `<form>` tag will be included in the data of the form submission event.

A couple important things to note when defining your HTML form:

1. You must set the `method` attribute of the form to `"post"`

2. If your HTML is specified in an external file/URL, as opposed to embedded in your script, you must set the `action` attribute to the vizhtml web server (e.g. "http://localhost:8080/vizhtml_form/")

For example, if you want to display a form contained in an external file, your form tag should look something like the following:

```
<form method="post" action="http://localhost:8080/vizhtml_form/">

.

.

.

</form>
```

Example

The following example displays an HTML form in Vizard and collects the form data once it's submitted:

```
import viz

import viztask

import vizhtml



viz.go()



def FormTask():



    html = """

    <html>
```

```
<body onload="document.the_form.first_name.focus();">


<form name="the_form" method="post">
    <table>
        <tr>
            <td>First Name</td>
            <td><input type="text" name="first_name"></td>
        </tr>
        <tr>
            <td>Last Name</td>
            <td><input type="text" name="last_name"></td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" value="Submit"></td>
        </tr>
    </table>
</form>


</body>
</html>"""


# Display input form
```

```
    vizhtml.displayCode(html)



    # Wait for form submission

    d = yield vizhtml.waitFormSubmit()



    # Get data from form

    first = d.first_name

    last = d.last_name



    # Display name

    import vizinfo

    vizinfo.InfoPanel('Hello {} {}'.format(first,last))

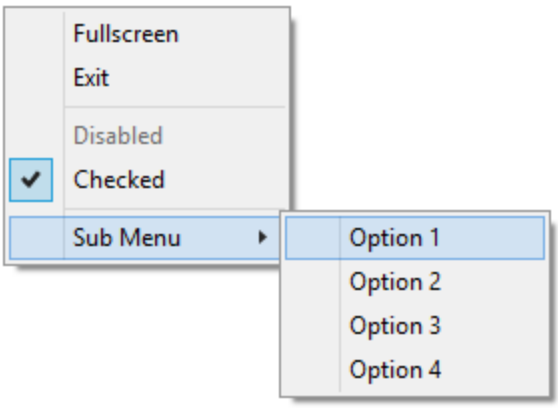


viztask.schedule( FormTask() )
```

vizpopup

Use the `vizpopup` module to create Windows style pop-up and drop down menus. Pop-up menus can be set to display with a right mouse click, as they are in Windows, or with any other event. Drop down menus are added to a menu bar at the top of the Vizard graphics window.

The same commands are used to create both types of menus and handle menu selection events. Items added to a menu can be enabled or disabled (grayed out) and checked or unchecked.



The following example shows how to create a pop-up menu and handle menu selection events:

```
import viz

import vizpopup



viz.go()



fullscreenItem = vizpopup.Item('Fullscreen')

exitItem = vizpopup.Item('Exit')

disabledItem = vizpopup.Item('Disabled',enabled=False)

checkedItem = vizpopup.Item('Checked',checked=True)



submenu = vizpopup.Menu('Sub Menu',[ vizpopup.Item('Option '+str(i)) for i in
range(1,5) ] )

mymenu =
vizpopup.Menu('Main',[fullscreenItem,exitItem,vizpopup.Separator,disabledItem,c
heckedItem,vizpopup.Separator,submenu])



def showMenu():
```

```
    fullscreenItem.checked = viz.window.getFullscreen()

    vizpopup.display(mymenu)


vizact.onmouseup(viz.MOUSEBUTTON_RIGHT,showMenu)


def onSelect(e):

    if e.menu is mymenu:

        viz.logNotice(e.item.name,' selected')

viz.callback(vizpopup.MENU_SELECT_EVENT,onSelect)


vizpopup.onMenuItem(fullscreenItem,viz.window.setFullscreen,viz.TOGGLE)

vizpopup.onMenuItem(exitItem,viz.quit)
```

Creating a Menu

`vizpopup` `Menus` are made up of `Items` (menu options), `Separators` (menu partitions), and other `Menus`.

The following command creates a `Menu` object:

Command	Description

vizpopup.Menu(name, items checked=None, enabled=True)	Creates a Menu object. If it becomes a submenu, the **name** will be displayed in the parent menu. **items:** A list of items that appear in the menu. The list can contain **vizpopup.Item**, **vizpopup.Separator**, and other **vizpopup.Menu** objects. The order of items in the list determines their order in the menu. **checked:** This applies if the menu becomes a submenu. When set to True a check mark will appear to the left of the submenu name in the parent menu. This can be modified after the object is created using the **<Menu>.checked** attribute. **enabled:** This applies if the menu becomes a submenu. When set to False, the submenu name will be grayed out in the parent menu. This can be modified after the object is created using the **<Menu>.enabled** attribute.

The following command creates an `Item`, an option in the `Menu` that can be selected:

Command	Description

vizpopup.Item(name, checked=None, enabled=True)	Creates an Item object. The string value passed in the **name** argument will display in the menu this item is added to. **checked:** When set to True a check mark will appear in the menu to the left of the Item name. This can be modified after the object is created using the **<Item>.checked** attribute. **enabled:** This determines whether the Item can be selected. If set to False, the Item name will be grayed out in the menu. This can be modified after the object is created using the **<Item>.enabled** attribute.

The following command creates a `Separator`, a line used to partition a `Menu`:

Command	Description
vizpopup.Separator	creates a Separator object.

Displaying a pop-up menu

The following commands control pop-up menu display:

Command	Description
vizpopup.display(menu)	Displays the specified menu.
vizpopup.cancel()	Hides the current menu.

Any event can be used to show the menu. Simply call the `display` command in the event's callback function. In the line of code below the right mouse button is used to show `myMenu`.

```
vizact.onmouseup(viz.MOUSEBUTTON_RIGHT,vizpopup.display,myMenu)
```

A menu that is displayed will automatically be hidden once the user makes a selection, the left/right mouse button is clicked, or the Esc/Alt key is pressed. When it is necessary to use some other event to hide the menu the `cancel` command can be called manually (e.g. a joystick button press).

> Keyboard events are disabled and cannot be used when a menu is active.

Setting the menu bar

Use the following command to set the menu bar at the top of the Vizard graphics window:

Command	Description
vizpopup.setMenuBar(menu)	Sets the menu bar for the graphics window.

menu: a **vizpopup.Menu** object.

The table below describes how items contained within the `Menu` object are displayed in the menu bar:

Object	Description
vizpopup.Menu	A drop down menu.
vizpopup.Item	A selectable item.
vizpopup.Separator	An empty space.

The menu bar can be modified anytime by calling the `setMenuBar` command and passing in a new `Menu` object. To remove the menu bar call the command and pass in a value of `None`.

The following example code shows how to add a menu bar with drop down menus:

```
import viz

import vizpopup

viz.go()



piazza = viz.addChild('piazza.osgb', scene=viz.Scene1)

dojo = viz.addChild('dojo.osgb', scene=viz.Scene2)
```

```
gallery = viz.addChild('gallery.osgb', scene=viz.Scene3)


music = viz.addAudio('bach_air.mid')

music.loop()


# Create scene menu items

piazzaItem = vizpopup.Item('Piazza')

dojoItem = vizpopup.Item('Dojo')

galleryItem = vizpopup.Item('Gallery')

sceneMenu = vizpopup.Menu('Scene',[piazzaItem,dojoItem,galleryItem])


#Create audio menu items

playItem = vizpopup.Item('Play')

pauseItem = vizpopup.Item('Pause')

audioMenu = vizpopup.Menu('Audio',[playItem,pauseItem])


# Create main menu bar

menuBar = vizpopup.Menu('MainMenu',[sceneMenu,audioMenu])

vizpopup.setMenuBar(menuBar)


# Create menu item actions

vizpopup.onMenuItem(piazzaItem, viz.MainWindow.setScene, viz.Scene1)

vizpopup.onMenuItem(dojoItem, viz.MainWindow.setScene, viz.Scene2)
```

```
vizpopup.onMenuItem(galleryItem, viz.MainWindow.setScene, viz.Scene3)

vizpopup.onMenuItem(playItem,music.play)

vizpopup.onMenuItem(pauseItem,music.pause)
```

Handling menu selection events

Menu selection events can be handled per Item using `vizpopup.onMenuItem()` or by registering a callback function for all menu selections using the `vizpopup.MENU_SELECT_EVENT`.

The `vizpopup.onMenuItem()` command handles an individual item selection and takes the following arguments:

Command	Description
vizpopup.onMenuItem(item, func, *args **kw)	Registers a callback function for the **item** specified. **func:** The function to call when the **item** is selected. ***args:** Arguments to pass to the function.

	****kw:** Keyword arguments to pass to the function.

The following line of code toggles between fullscreen and windowed mode when the `fullscreenItem` is selected:

```
vizpopup.onMenuItem(fullscreenItem,viz.window.setFullscreen,viz.TOGGLE)
```

The `vizpopup.MENU_SELECT_EVENT` event is generated when any item is selected. It provides a single event structure **e** with the following attributes:

Attribute	Description
e.menu	The menu object that generated the event.
e.item	The item object that generated the event.

In the code below, if an item from `myMenu` is selected it's name will be printed.:

```
def onSelect(e):
    if e.menu is myMenu:
        viz.logNotice(e.item.name,' selected')
viz.callback(vizpopup.MENU_SELECT_EVENT,onSelect)
```

viztip

The `viztip` library allows you to assign a `tooltip` attribute to any node object that contains the `tooltip` text. Once you create an instance of the `viztip.ToolTip` object, it will

automatically display an on-screen tooltip when the mouse is over the node. The following example demonstrates this:

```
import viz

viz.go()


viz.add('tut_ground.wrl')


#Create models with tooltip text

ball1 = viz.add('ball.wrl',pos=(-2,1,5))

ball1.tooltip = 'Ball 1'


ball2 = viz.add('ball.wrl',pos=(0,1,5))

ball2.tooltip = 'Ball 2'


ball3 = viz.add('ball.wrl',pos=(2,1,5))

ball3.tooltip = 'Ball 3'


#Create a tool tip object

import viztip

tip = viztip.ToolTip()
```

Methods and Events

A number of methods are available on the `Tooltip` object. These control properties such as the time for the `Tooltip` to appear, whether it tracks the mouse position, it's visibility, and

more. Events generated after a `Tooltip` object have been created include the `viztip.TOOLTIP_BEGIN_EVENT` and `viztip.TOOLTIP_END_EVENT`:

```
import viz

viz.go()


gallery = viz.add('gallery.ive')


#Create models with tooltip text

ball = viz.add('ball.wrl',pos=(0,1.8,2))

ball.tooltip = 'Ball'


#Create a tool tip object

import viztip

tip = viztip.ToolTip()


#set initial tooltip message and position

tip.message('Move the cursor over the ball')

tip.translate(300,50)


#action to add when mouse is over a tooltip object

spin = vizact.spin(0,1,0,45)


#handle toolpit events
```

```
def beginTip(e):

    e.node.addAction(spin)



viz.callback(viztip.TOOLTIP_BEGIN_EVENT,beginTip)



def endTip(e):

    e.node.endAction()



viz.callback(viztip.TOOLTIP_END_EVENT,endTip)
```

vizdragdrop

The `vizdragdrop` module allows you to drag and drop files on the Vizard graphics window. It supports dropping one or more files at the same time. When a file is dropped an event is generated that provides the pathname of the file and the mouse position to the registered callback function. This can used as a way to add models and other types of resources to the scene at runtime.

To see how this works, run the following script and drop the `vcc_male.cfg` file (found in the `resources` folder of the Vizard installation) on the graphics window:

```
import viz

import vizdragdrop



viz.go()



court = viz.addChild('court.ive')
```

```
model_path = viz.res.getFullPath('vcc_male.cfg')


vizdragdrop.enable()


def onDropModelFile(e):


    if e.filenames[0] == model_path:

        male = viz.addAvatar('vcc_male.cfg',pos = [0,0,5],euler = [180,0,0])

        male.state(1)


vizdragdrop.onDropFile('*.cfg',onDropModelFile)
```

vizdragdrop commands

The following commands enable and disable file drag/drop:

Command	Description
vizdragdrop.enable()	Enables file drag/drop on the Vizard window.
vizdragdrop.disable()	Disables file drag/drop on the Vizard window.

When drap/drop is enabled and a file is dragged on the graphics window a drag/drop marker with "+" sign will appear that follows the mouse cursor. Once the file is dropped a `vizdragdrop.DROP_FILE_EVENT` is generated. Use the following command for registering a callback function to handle this event:

Command	Description
vizdragdrop.onDropFile(pattern, func, *args, **kw)	Registers the function that gets called when a filename matching the specified **pattern** is dropped. If **pattern** is None, then all dropped files will be passed to the function. **func**: The function being registered. An event object **e**, described in the table below, will be passed to this callback function. ***args**: Optional arguments to pass to the registered function. ****kw**: Optional keyword arguments to pass to the registered function.

The drop event provides a single event object e that gets passed to the callback function. The table below lists the object's attributes:

Attribute	Description

e.filenames	A list that holds the pathname(s) of the file(s) dropped
e.pos_normalized	Mouse position in normalized window coordinates
e.pos_pixels	Mouse position in pixel window coordinates
e.pos	Same as e.pos_normalized

Example

The following example uses the mouse position at drop time to determine where an avatar should be placed. Run the script and individually drop the `vcc_male.cfg` and `vcc_female.cfg` files on the graphics window (These files can be found in the resources folder of the Vizard installation). To change the position of an avatar drop it's .cfg file again in a new location:

```
import viz

import vizdragdrop



viz.go()



court = viz.addChild('court.ive')
```

```
male = viz.addAvatar('vcc_male.cfg',euler = [180,0,0])

male.state(5)

male.visible(viz.OFF)



female = viz.addAvatar('vcc_female.cfg',euler = [180,0,0])

female.state(5)

female.visible(viz.OFF)



#Get the full paths to the avatar cfg files to check

#against in the onDropModelFile function

male_path = viz.res.getFullPath('vcc_male.cfg')

female_path = viz.res.getFullPath('vcc_female.cfg')



#Turn off mouse navigation

viz.mouse(viz.OFF)



#Enable file drag/drop

vizdragdrop.enable()



def onDropModelFile(e):



    #Determine if the file dropped is either vcc_male.cfg or vcc_female.cfg

    if e.filenames[0] == male_path:
```

```
        avatar = male
    elif e.filenames[0] == female_path:
        avatar = female
    else:
        return

    #Get the mouse position at drop time and use that to
    #calculate the avatar position
    mouse_pos = e.pos_normalized
    x_pos = 5 * mouse_pos[0] - 2.5

    #make the avatar visible and set it's position
    avatar.visible()
    avatar.setPosition([x_pos,0,5])

vizdragdrop.onDropFile('*.cfg',onDropModelFile)
```

GUI command table

viz module commands for adding GUI elements

<viz>.addButtonLabel
<viz>.addCheckbox
<viz>.addButton
<viz>.addButtonLabel
<viz>.addDropList
<viz>.addRadioButton

<viz>.addSlider
<viz>.addText
<viz>.addTextbox
<viz>.setTheme

node3d commands for working with GUIs

> **Note:** GUI elements inherit generic node3d commands as well

<node3d:GUI>.disable
<node3d:GUI>.enable
<node3d:GUI>.getFocus
<node3d:GUI>.getLength
<node3d:GUI>.setFocus
<node3d:GUI>.setLength
<node3d:GUI>.setTheme

<node3d:GUI:button>.downpicture
<node3d:GUI:button>.get
<node3d:GUI:button>.set
<node3d:GUI:button>.uppicture

<node3d:GUI:buttonlabel>.get
<node3d:GUI:buttonlabel>.overflow

<node3d:GUI:droplist>.addItem
<node3d:GUI:droplist>.addItems
<node3d:GUI:droplist>.clearItems
<node3d:GUI:droplist>.getItem
<node3d:GUI:droplist>.getItemCount
<node3d:GUI:droplist>.getItems
<node3d:GUI:droplist>.getSelection
<node3d:GUI:droplist>.overflow
<node3d:GUI:droplist>.removeItem

<node3d:GUI:droplist>.select

<node3d:GUI:progressbar>.get
<node3d:GUI:progressbar>.set

<node3d:GUI:slider>.barpicture
<node3d:GUI:slider>.dragging
<node3d:GUI:slider>.get
<node3d:GUI:slider>.horizontal
<node3d:GUI:slider>.margin
<node3d:GUI:slider>.set
<node3d:GUI:slider>.setTickSize
<node3d:GUI:slider>.tickpicture
<node3d:GUI:slider>.vertical

<node3d:GUI:textbox>.getCursorPos
<node3d:GUI:textbox>.overflow
<node3d:GUI:textbox>.setCursorPos

Actions for GUI events

<vizact>.onbuttondown
<vizact>.onbuttonup
<vizact>.onlist
<vizact>.onslider

Task conditions for GUI events

<viztask>.waitButtonDown
<viztask>.waitButtonUp

HTML forms in a remote browser

In order to collect data from an HTML form displayed in a remote browser, a page containing HTML content needs to be registered with `vizhtml's` local HTTP server. A form submission

event will get triggered once a form is submitted and data can be collected using a task or callback function.

The following commands register pages with HTML content:

Command	Description
vizhtml.registerFile(url, filename, directory=None)	Registers a page with the vizhtml server that uses an HTML file for content. **url:** The path portion of the page's URL. See the server and URL section for further details about URLs. **filename:** The name of the HTML file. **directory:** The directory where page resources are saved.

vizhtml.registerCode(url, code, directory=None)	Registers a page with the vizhtml server that uses HTML code for content. **url:** The path portion of the page's URL. See the server and URL section for further details about URLs. **code:** A string that contains HTML code. **directory:** The directory where page resources are saved.

In the following example, HTML code that contains a form is registered with the server. Once the form is submitted from a browser, it's data is collected in Vizard.

For testing on a local browser use 'localhost' or the loopback address:

http://localhost:8080/vizhtml/form_submit
http://127.0.0.1:8080/vizhtml/form_submit

For remote browsers replace computer name or IP address with the Vizard machine's name or address

http://computer name:8080/vizhtml/form_submit
http://IP Address:8080/vizhtml/form_submit

```
import viz
```

```
import vizhtml



viz.go()



code = """

<html>

<head>

    <title>vizhtml Form Submit Example</title>

</head>

<body>

<form name="the_form" method="post">

            <table>

                <tr>

                    <td>First Name</td>

                    <td><input type="text" name="first_name"></td>

                </tr>

                <tr>

                    <td>Last Name</td>

                    <td><input type="text" name="last_name"></td>

                </tr>

                <tr>

                    <td></td>

                    <td><input type="submit" value="Submit"></td>
```

```
                </tr>

            </table>

        </form>

</body>

</html>

"""


vizhtml.registerCode('form_submit',code)


def HandleForm(e):

    print(e.first_name,e.last_name)

vizhtml.onFormSubmit(HandleForm)
```

Custom handlers

When an HTTP request (e.g. page request, resource request, form submission) arrives at `vizhtml's` HTTP server, it get's handled so the appropriate action is taken.

By registering your own handler with the server you can have perform custom actions when a request arrives. The handler will be passed a request object that provides information about the browser's request and methods to send content back to the browser.

The following command registers a handler with the server:

Command	Description

vizhtml.registerHandlerClass(url, handler_class directory=None)	Registers a custom handler class with the vizhtml server. **url:** The path portion of the URL. See the server and URL section for further details about URLs. **handler_class:** A custom handler class that extends the vizhtml.PageRequestHandler class. **directory:** The directory where page resources are saved.

The custom handler should inherit from `vizhtml.PageRequestHandler` and override it's `handle_request` method. This method is passed a request object that is an instance of `vizhtml.VizardHTTPRequestHandler`.

Note: The handle_request method is called from a separate thread than the main thread.

`VizardHTTPRequestHandler` inherits from Python's `BaseHTTPRequestHandler` class and adds the following methods and attributes:

Attribute	Description

<VizardHTTPRequestHandler>.form_event	The form event if the request is from a form submission, otherwise None.
<VizardHTTPRequestHandler>.resource_path	The relative path of a resource request from custom page (e.g. images).

Method	Description
<VizardHTTPRequestHandler>.send_html_code(code)	Sends the specified HTML code to the browser.
<VizardHTTPRequestHandler>.send_html_file(filename)	Sends the specified HTML file to the browser.
<VizardHTTPRequestHandler>.send_local_resource(filename, mimetype=None)	Sends the specified resource file to the browser. **mimetype:** If this is None it will be guessed from the **filename**. Otherwise it must be a 2-item tuple containing the HTTP content type and content encoding strings.

The example code below uses a custom handler to update content on a web page after a form submission. The data retrieved from the form's text field is placed back into the page and Vizard's frame count is displayed.

For testing on a local browser use 'localhost' or the loopback address:

http://localhost:8080/vizhtml/custom_handler
http://127.0.0.1:8080/vizhtml/custom_handler

For remote browsers replace computer name or IP address with the Vizard machine's name or address

http://computer name:8080/vizhtml/custom_handler
http://IP Address:8080/vizhtml/custom_handler

```
import cgi

import viz

import vizinfo

import vizhtml



viz.go()



# Display form submitted message on screen

info = vizinfo.InfoPanel('')



code = """

<html>

<head>

    <title>vizhtml Custom Handler Example</title>
```

```
</head>

<body onload="document.the_form.message.focus();">

<img src='ball.jpg'>

<form name="the_form" method="post">

    <table>

        <tr>

            <td>Frame Number:</td>

            <td>{frame}</td>

        </tr>

        <tr>

            <td>Message:</td>

            <td><input type="text" name="message" value="{message}"></td>

        </tr>

        <tr>

            <td></td>

            <td><input type="submit" value="Submit"></td>

        </tr>

    </table>

</form>

</body>

</html>

"""
```

```
class ExampleRequestHandler(vizhtml.PageRequestHandler):



    def handle_request(self,request):

        """@args vizhtml.VizardHTTPRequestHandler()"""



        # If request is for a resource (e.g. image), then let vizhtml handle
it

        if request.resource_path:

            return



        # If form was submitted, put message back into page

        if request.form_event:

            message = cgi.escape(request.form_event.message,True)

        else:

            message = ''



        # Send html code with dynamic content


request.send_html_code(code.format(frame=viz.getFrameNumber(),message=message))



vizhtml.registerHandlerClass('custom_handler',ExampleRequestHandler,directory='
.')



def HandleForm(e):

    info.setText(e.message)
```

```
vizhtml.onFormSubmit(HandleForm)
```

Collecting Form Data

When an HTML form in a remote browser or embedded HTML window is submitted, a `vizhtml.FORM_SUBMIT_EVENT` is generated. The event can be handled using either a task or callback function.

Waiting for a form submission in a viztask function

Task functions, from the `viztask` library, are useful for controlling program flow. Use the following command to create a `viztask` condition that waits for a form submission:

Command	Description
vizhtml.waitFormSubmit()	Waits until a form submit event occurs. The yielded command returns a viz.Data object that holds the form data.

The names of properties in the returned data object match the names of input fields in the HTML form. For example, if the HTML form contains a text field named `occupation`:

```
<input type="text" name="occupation">
```

the code below would store the value from that field after the form is submitted:

```
def myFormTask():


    # Display input form

    vizhtml.displayCode(html)
```

```
    # Wait for form submission
    d = yield vizhtml.waitFormSubmit()

    # Get data from form
    occupation = d.occupation

viztask.schedule( myFormTask() )
```

Handling a form submission in a callback function

Use the command below to register a callback function for a form submission event:

Command	Description
vizhtml.onFormSubmit(func, *args, **kw)	Registers a callback function for a form submission event: **func**: The function being registered. An event object **e**, holding form data, will be passed to this callback function. ***args**: Optional arguments to pass to the registered function.

	****kw**: Optional keyword arguments to pass to the registered function.

The names of properties in the event object **e** passed to the callback function match the names of input fields in the HTML form. For example, if the HTML form contains a text field named `occupation`:

```
<input type="text" name="occupation">
```

the code below would store the value from that field after the form is submitted:

```
# Display input form

vizhtml.displayCode(html)



def HandleForm(e):



    occupation = e.occupation



vizhtml.onFormSubmit(HandleForm)
```

Local HTTP server and URLs

All the communication that occurs between Vizard and a browser or embedded HTML page, whether through forms or WebSocket messages, goes through a local HTTP server built into `vizhtml`. The commands listed in previous sections to display HTML in the embedded window and register HTML with the local HTTP server automatically start the server. The following table lists commands to manually start and stop the server and get it's base URL:

Command	Description
vizhtml.startServer()	Starts the local HTTP server.
vizhtml.stopServer()	Stops the local HTTP server.
vizhtml.getServerURL()	Gets the base URL of the local HTTP server.

The base URL of `vizhtml's` HTTP server is the default server port (8080) affixed to the computer name or IP address of the machine. The port can be modified by setting the `vizhtml.port` option before calling any `vizhtml` commands. See the Global settings page for information on setting options.

The full URL of a page on the server is a combination of the base URL of the server + /vizhtml/ + the page URL. For example, if the Vizard program is run on a computer with name 'Vizard-Renderer' and IP address '192.168.0.5' and the following line is used to register HTML code:

```
vizhtml.registerCode('form_submit',code)
```

then **http://Vizard-Renderer:8080/vizhtml/form_submit** or **http://192.168.0.5:8080/vizhtml/form_submit** should be entered into the URL window of a remote browser to display the page.

WebSocket Communication

Real time bidirectional communication between Vizard and a web browser is supported using WebSockets. You can send and receive messages and find out when socket connections open or close. On the server (Vizard) side, the vizhtml module provides Vizard commands for sending messages and handling message and connection events. Similar functionality is

provided on the client (browser) side using JavaScript functions through vizhtml's JavaScript library.

All the major web browsers support WebSockets. The minimum supported version for each browser is:

- Chrome 14
- Firefox 6
- Internet Explorer 10
- Safari 6

Vizard commands

Use the following commands to send WebSocket messages and get information about connected clients:

Command	Description
vizhtml.sendAll(event, data)	Sends WebSocket data to all connected clients. **event:** A string value that defines the event name. **data:** Any basic object type (e.g. int, string , list, ...) or a viz.Data object.

vizhtml.getClientList()	Gets a list containing the currently connected clients. Each element in the list is an instance of the **vizhtml.WebSocketClient** class.
vizhtml.getClientCount()	Returns the number of currently connected clients.

Sending data to a specific client is accomplished through it's associated `vizhtml.WebSocketClient` object. You can get a handle to this object when a WebSocket event occurs (e.g. connection event) or by using the `vizhtml.getClientList()` command. Each object provides the client's IP address and a method to send it data:

Attribute	Description
<WebSocketClient>.address	The IP address of the client.

Method	Description
<WebSocketClient>.send(event, data)	Sends WebSocket data to the client. **event:** A string value that defines the event name.

	data: Any basic object type (e.g. int, string , list, ...) or a viz.Data object.

Handling events

The following events will get triggered when messages are received and connections are opened and closed:

Event	Description
vizhtml.MESSAGE_EVENT	Generated when a WebSocket message is received. It provides a single event structure **e** with the following attributes: **e.event:** A string with the event name. This should match an event_name defined in a **<VizardWebSocket>.send** command (described in the JavaScript section below). **e.data:** The data sent. **e.client:** A **vizhtml.WebSocketClient** object.

vizhtml.CONNECT_EVENT	Generated when a WebSocket client connects. It provides a single event structure **e** with the following attribute: **e.client:** A **vizhtml.WebSocketClient** object.
vizhtml.DISCONNECT_EVENT	Generated when a WebSocket client disconnects. It provides a single event structure **e** with the following attribute: **e.client:** A **vizhtml.WebSocketClient** object.

Use the following commands to register callback functions for these events:

Command	Description
vizhtml.onMessage(event, func, *args, **kw)	Registers a callback function for a message event: **event:** A string with the event name. This should match an **event_name** defined in a

	<VizardWebSocket>.send command (described in the JavaScript section below). **func**: The function being registered. An event object **e,** described in the table above (see **vizhtml.MESSAGE_EVENT**) will be passed to the callback function. ***args**: Optional arguments to pass to the registered function. ****kw**: Optional keyword arguments to pass to the registered function.
vizhtml.onConnect(func, *args, **kw)	Registers a callback function for a WebSocket connection event: **func**: The function being registered. An event object **e,** described in the table above (see **vizhtml.CONNECT_EVENT**) will be passed to the callback function.

	***args**: Optional arguments to pass to the registered function. ****kw**: Optional keyword arguments to pass to the registered function.
vizhtml.onDisconnect(func, *args, **kw)	Registers a callback function for WebSocket disconnection event: **func**: The function being registered. An event object **e,** described in the table above (see **vizhtml.DISCONNECT_EVENT**) will be passed to the callback function. ***args**: Optional arguments to pass to the registered function. ****kw**: Optional keyword arguments to pass to the registered function.

Javascipt commands

Use `vizhtml's` JavaScript library to create a WebSocket client that communicates with `vizhtml's` HTTP server. In order to access the library within your own JavaScript code, link to `vizhtml.js` as an external script:

```
<script type="text/javascript" src="vizhtml.js"></script>
```

To implement a WebSocket client create an instance of the `viz.WebSocket` class. Before creating a WebSocket, you can check whether the browser supports WebSockets by calling the `viz.isWebSocketSupported()` command. This provides a way to display a helpful error message to the user.

The following methods are available on the `viz.WebSocket` object:

Method	Description
<WebSocket>.onevent(event_name, func)	Registers a callback function for an event. **event_name:** A string with the event name. For message events, this argument should match an **event** value defined in one of the server commands (either **vizhtml.sendAll** or **<WebSocketClient>.send**). Otherwise, see the table below for connection event values.

	func: The JavaScript function being registered. An event object **e** with the following attributes will be passed to the callback function: • **e.event:** A string with the event name. This is the same as the **event_name** argument. • **e.data:** The event data.
<WebSocket>.send(event_name, event_data)	Sends data to the WebSocket server. **event_name:** A string value that defines the event name. **event_data:** Any basic object type (e.g. int, string , array, ...) or the return object from the `viz.Data` function.

The table below lists `event_name` values for connection events,

Value
'open'

'close'
'error'

Sending a viz.Data object to Vizard

If you want the JavaScript code to send an object that appears to Vizard as a `viz.Data` object, then you can use the `viz.Data` JavaScript function:

Command	Description
viz.Data(data)	Returns an object that will be converted to a `viz.Data` object when sent to the vizhtml server. **data:** a JavaScript associative array that uses strings as keys.

In the following JavaScript code, the integer value stored in `click_count` is sent as the `count` attribute of a `viz.Data` object:

```
var click_count = 0;

function onButtonClick()

{

    click_count += 1;
```

```
    socket.send( 'button_click' , viz.Data({'count':click_count}) );

}
```

In Vizard, the following code retrieves and prints the value:

```
def OnButtonClick(e):

    data = e.data

    print('BUTTON CLICK',data.count)

vizhtml.onMessage('button_click',OnButtonClick)
```

Example

The following example script shows how to communicate between Vizard and a browser via WebSockets.

Note: If no communication occurs it's likely your browser does not support WebSockets.

For testing on a local browser use 'localhost' or the loopback address:

http://localhost:8080/vizhtml/websocket
http://127.0.0.1:8080/vizhtml/websocket

For remote browsers replace computer name or IP address with the Vizard machine's name or address

http://computer name:8080/vizhtml/websocket
http://IP Address:8080/vizhtml/websocket

```
import viz

import vizhtml

viz.go()
```

```
code = """

<html>

<head>

    <title>vizhtml WebSocket Example</title>

    <script type="text/javascript" src="vizhtml.js"></script>

</head>

<body>

<script language="javascript">



var socket = new viz.WebSocket();



socket.onevent('open', function(e){

    document.getElementById('status').innerHTML = 'connected';

    document.getElementById('my_button').disabled = false;

})



socket.onevent('close', function(e){

    document.getElementById('status').innerHTML = 'waiting for connection';

    document.getElementById('my_button').disabled = true;

    socket.reconnect();

})



socket.onevent('set_time', function(e){
```

```
    document.getElementById('script_time').innerHTML = e.data.time;

})



var click_count = 0;

function onButtonClick()

{

    click_count += 1;

    socket.send( 'button_click' , viz.Data({'count':click_count}) );

}



</script>



<div>Status:</div><div id='status'>waiting for connection</div></br>

<div>Time:</div><div id='script_time'></div></br>

<div><input id='my_button' type=button value="Send Message"
onclick="onButtonClick();"></div></br>

</body>

</html>

"""



# http://localhost:8080/vizhtml/websocket

vizhtml.registerCode('websocket',code)



def ClientConnect(e):
```

```
    print('Client connected at',e.client.address)

vizhtml.onConnect(ClientConnect)



def ClientDisconnect(e):

    print('Client disconnected at',e.client.address)

vizhtml.onDisconnect(ClientDisconnect)



def SendTime():

    d = viz.Data(time='%.1f'%(viz.tick()))

    vizhtml.sendAll('set_time',d)

vizact.ontimer(0.1,SendTime)



def OnButtonClick(e):

    print('BUTTON CLICK',e.data.count,'from',e.client.address)

vizhtml.onMessage('button_click',OnButtonClick)
```