Creating/Editing a Titan-Like Gui

This guide is for people who are interested in creating a GUI for a command line tool.

Using Titan's Java Framework

For this part of the guide. We assume that you have access to the code for either Titan or Puffin, and are planning on using it to create a similar gui.

Alternatively, this part of the guide also shows how to modify the GUIs for Puffin or Titan (since Puffin was built off Titan). Note that in some places there a series of xs as to specify the name of the program (since this applies to both Titan and Puffin).

Code structure

The code is broken up into four folders:

graphics – The code for the different entry boxes we can add into our code. To add a different kind of entry box, you will probably have to copy and modify one of this files.

io – Handles how data is stored and collected. You probably won't need to modify this code.

jobs – For code related to running jobs, creating files, and other such activities.

gui – Stores the code that makes up the gui, importing from io and graphics. If you want to add tabs or add lines to tabs, the files you want to edit are here.

How it works

When a job is ran, the following things happen:

1. All the information from the tabs is collected into a namePairValueGroup. (Using a subclass from the main class.)

2. We check to see if any needed information is missing (using the checkRequired class from jobs, which checks for the files specified in xxxGUIConstants).

3. We create an output folder if we haven't created one for this run already (using makeDirectory class from jobs).

4. We create a shell script of the command we want to run (using the makeFiles class from jobs).

5. We create a process to run the shell script without freezing the gui (using the runProcess class from jobs).

6. Our process can update various displays while running and upon completion.

Adding an new input to the Gui

Suppose that your backend code has recently added a new feature that allows putting in a date. You want to ask the user to input a date.

1. In the class xxxGUIConstants, add the name of your field. Also in xxxGUIConstants you can specify whether you want this field to default to something when the program starts up, and whether it is required for the run.

2. Choose one of the classes from the “graphics” folder as the way to input the date. You will most likely want a textInput for something like a date. However, some entries will work better with checkboxes, filesearches, or a custom made input class.

3. In the class for the tab you want to add the entry on. Add the entry class as a private variable. Increase the size of the values array.

4. Initiate your entry class. Add it to the values array. Then add it to the panel (with the add function).

5. Edit the makeFiles class to include your option in the run shell script.

Adding a New Tab to the Gui

Suppose you want to add a whole new tab of entries to the Gui.

1. Extend the class mayaGuiTab to create a class for your tab. Add in all your inputs onto your tab.

2. In the main class, add a private variable for your tab. Then initiate your tab and add it. In the fetchData subclass, add in your class to the getData function.

Using a Python GUI Builder

If you do not want to use Java, you may want to use one of python’s

Options for GUI Builders

I spent a bit of time looking for a good python gui builder. Of the ones I found:

Boa Constructor – My tool of choice.

WXGlade – I found this tool rather unstable, it crashed several times on my machine.

Python TK Gui Builder – A good tool, but its output uses the TK python library instead of the wxpython library.

Running a job

I created a small python module to help with running jobs.

From what I've seen of the GUIs, they help you with the layout and give you space to implement what happens on button presses.

So here is how you use my class:

1. Import it. (import runJob)

runJob.py (1 KB, uploaded by Alejandro Uriel Carbonara 1 decade 1 year ago)

2. In order to receive feedback from a running job, you need to implement two functions.

addText(self, text) This gives what your job would normally be outputting to the command line. Consider adding it to a noneditable text box.

updateGui(self, state, job): This function is called twice, once when the job starts and once when the job ends. You provide a job name when you initiate the function. The state will be "Start" when the job starts and "Finished" when the job finishes. If you want to make a button unusable while your job is running, use this.

3. If you want to run in a folder, use os.chdir() to navigate to the folder you want to run in.

4. To run your job, first create a new runJob class. It takes 3 arguments:

A list whose first element is the function you want to run, and whose remaining elements are the arguments to that function. So if you want to run "ls -al ./subfolder" you submit the list "-al", "./subfolder" .

Second a name for your job. This will be sent back to you in the updateGui function.

The final argument should just be "self", so that the thread can call the update functions.

5. Call the start function of your newly created class and you are done.

As an example, here is some code from a simple example app I made to check whether the module worked. Note that this is only a small part of the larger app, and that the name of the OnButton1Button function was generated by the Gui builder.

```
    def OnButton1Button(self, event):
        #Creating a runscript
        f = open("runScript.sh", "w")
        f.write("#!/bin/bash \n")
        f.write("touch ")
        f.write(self.textCtrl1.GetValue())
        f.close()
        os.chmod("runScript.sh", 0744)
        #Running the job
        t = runJob(["./runScript.sh"], "ExampleJob" , self)
        t.start()
        event.Skip()
        #Functions needed to display
    def addText(self, text):
        print text
    def updateGui(self, state, job):
        self.textCtrl2.SetValue(state)
        print state, job
```