Maya Batch Renderer GUI

The maya batch renderer is a command line tool to allow the rendering of frames from within a maya file. It has many command line options which can be determined by running the command Render -h. From this output the following elements have been identified as most use for the basic batch renderer dialog.

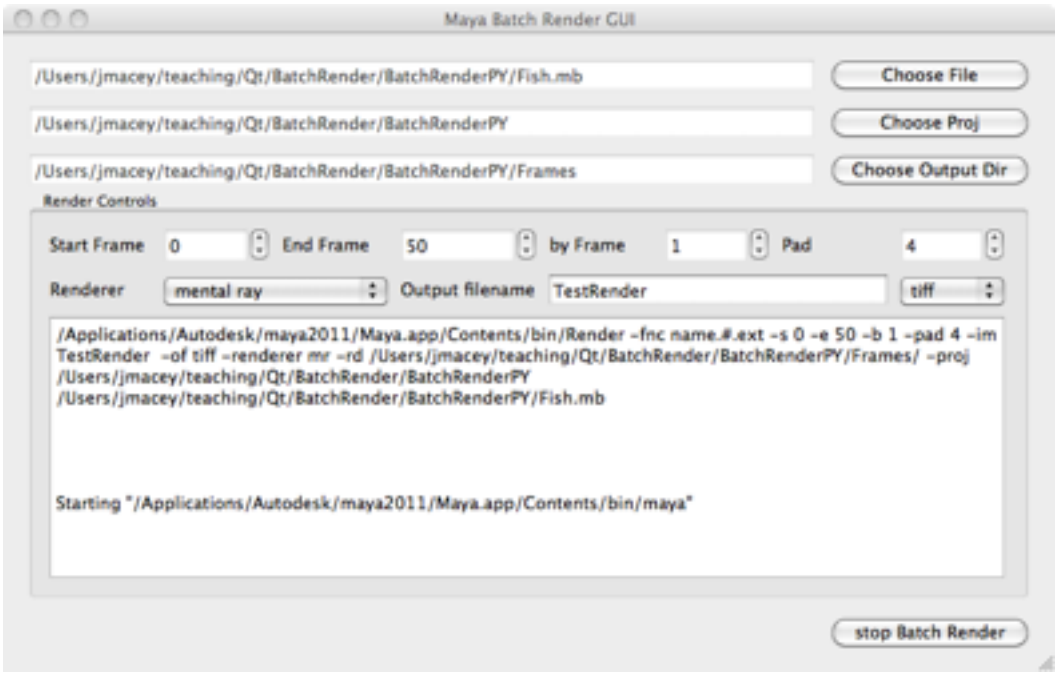
```
Usage: ./Render [options] filename
where "filename" is a Maya ASCII or a Maya binary file.

-renderer string   Use this specific renderer
-proj string       Use this Maya project to load the file

General purpose flags:
-rd path                        Directory in which to store image file
-of string                      Output image file format.
                                See the Render Settings window to
                                find available formats
-im filename                    Image file output name

Frame numbering options
-s float                        Starting frame for an animation sequence
-e float                        End frame for an animation sequence
-b float                        By frame (or step) for an animation sequence
-pad int                        Number of digits in the output image frame file name
                                extension
-fnc int                        File Name Convention: any of name, name.ext, ... See the
                                Render Settings window to find available options. Use
                                    namec and
                                namec.ext for Multi Frame Concatenated formats. As a
                                    shortcut,
                                numbers 1, 2, ... can also be used
```

In addition to this we can query the different renderer options and get the following list

render key	Usage
default	Use the renderer stored in the Maya file
file	Use the renderer stored in the Maya file
hw	Maya hardware renderer
mr	Mentalray renderer
rman	RenderMan renderer
sw	Maya software renderer

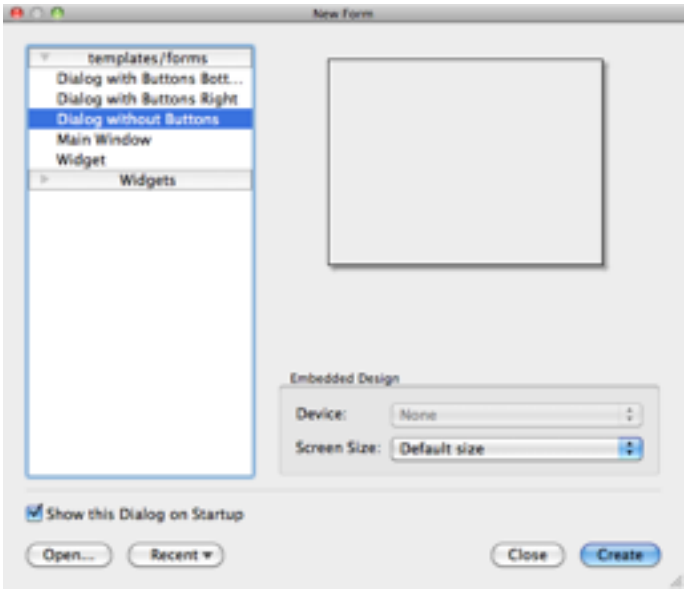
We are going to design a user interface using Qt and Python to generate the command line arguments shown above and give the user the ability to choose the files, project directory and output directory for the program.

The program will also report the output of the batch renderer in a window and give the user the ability to stop batch render at any stage. The main UI is shown next.



Batch Render Dialog

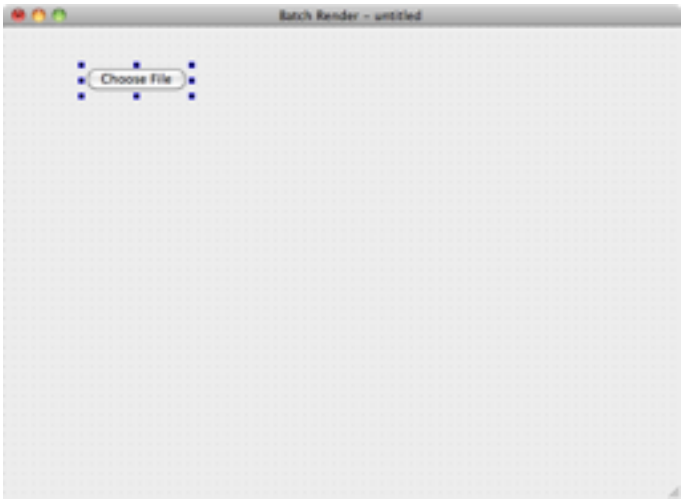
First open up designer (/opt/qtsdk/qt/bin/designer in the Linux studios) and choose a Dialog without buttons as shown



Select the dialog that's created and set the object properties objectName to mainDialog and windowTitle to Batch Render as shown

We are now going to add a button to the window and then set the layout manager before we create the rest of the UI.

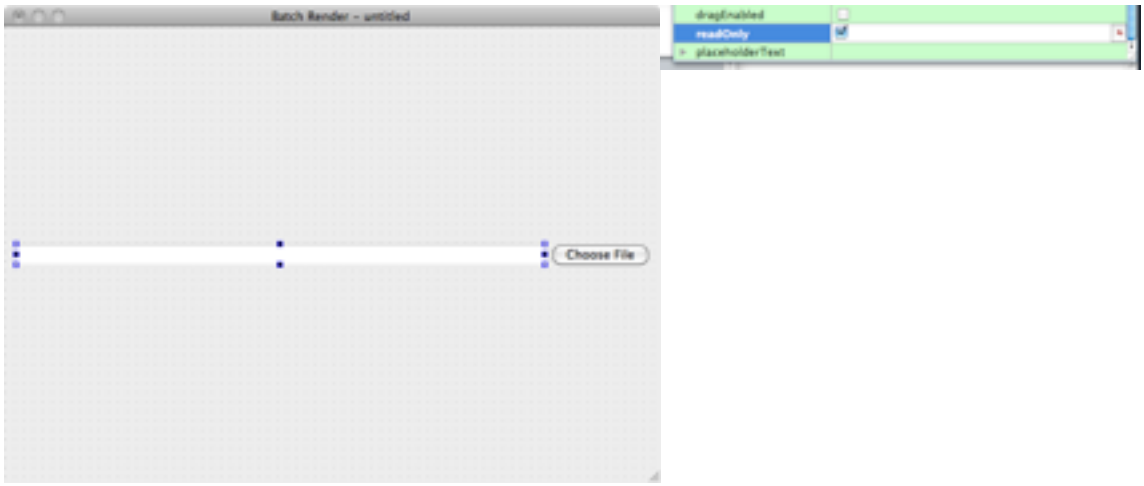
First drag a button anywhere on the screen, then change the name of the button to m_chooseFile and the button text to Choose File as shown below.



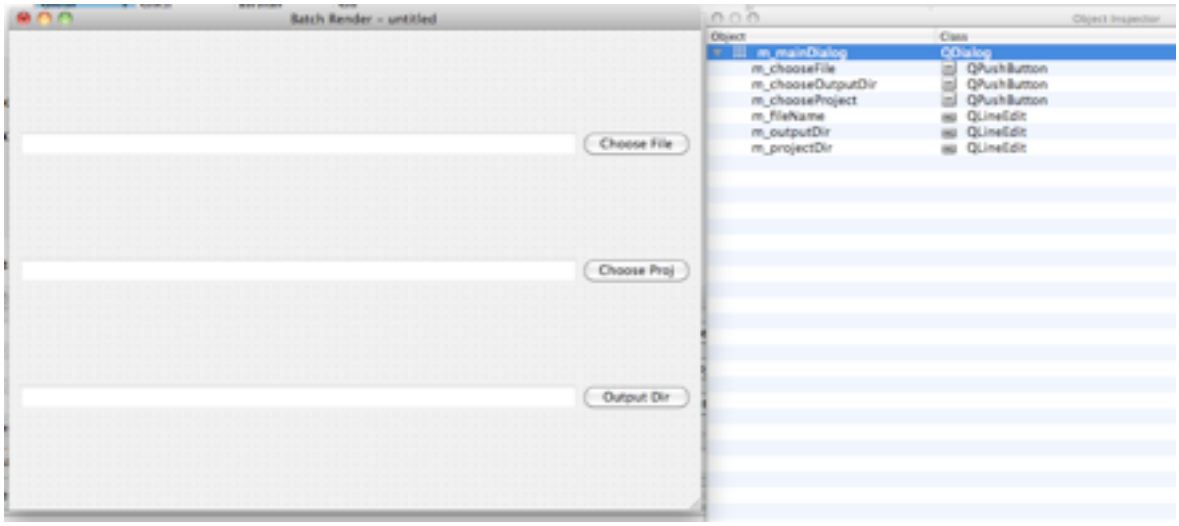
At present you are free to move any of the UI components within the form, however once the form is re-sized no of the buttons will re-size correctly. To enable this we need to add a

layout manager to the form. This is done by right clicking on the dialog and in this case we are going to select the “Layout on Grid” which should now result in the following

Now as we add components to the UI blue areas will appear as slots to add to the grid, for the next stage we are going to add a “QLineEdit” component next to the button, and name it m_fileName we will also tick the read-only tickbox.

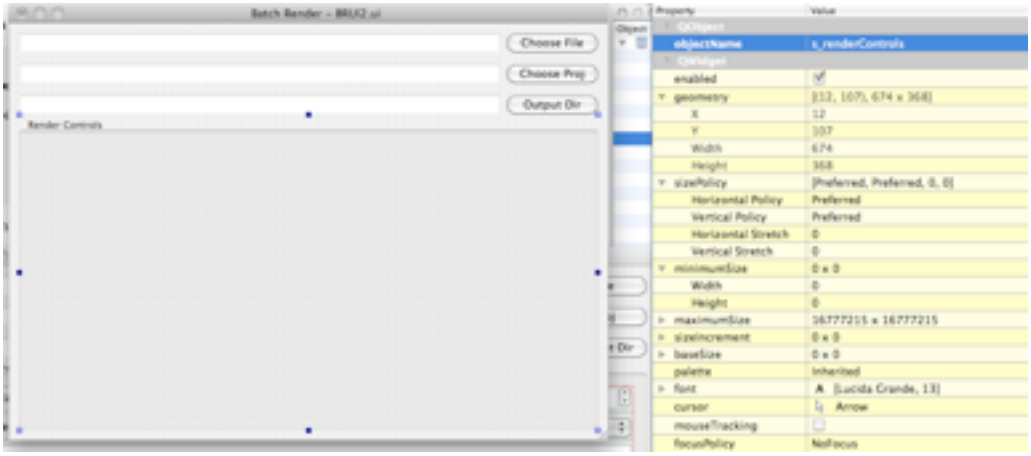


We are now going to replicate this process and add 2 more QLineEdit and Button Combinations as shown below

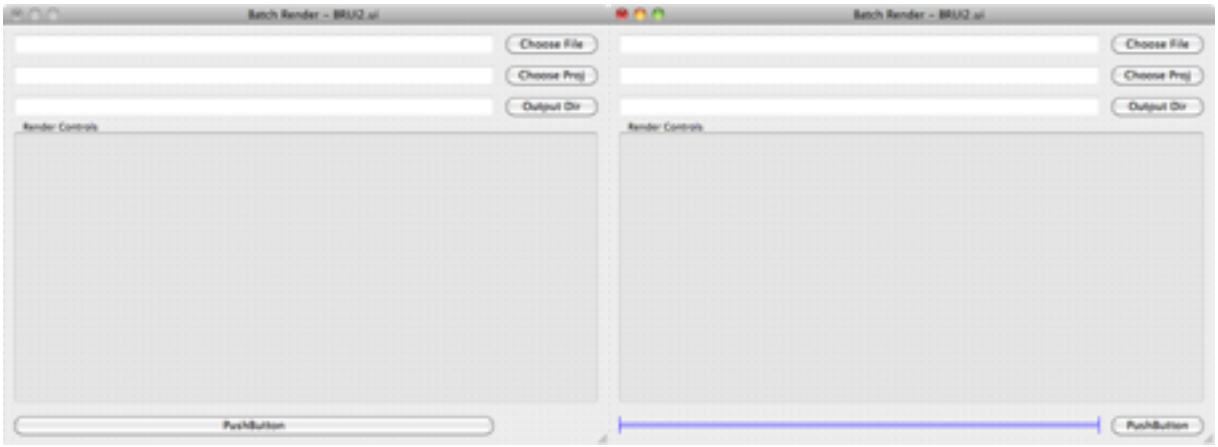


Note the Names of each of the components and set them to the correct names, and set the read only flag for each of the text components.

Next we are going to add a group box and set it to the following size and values

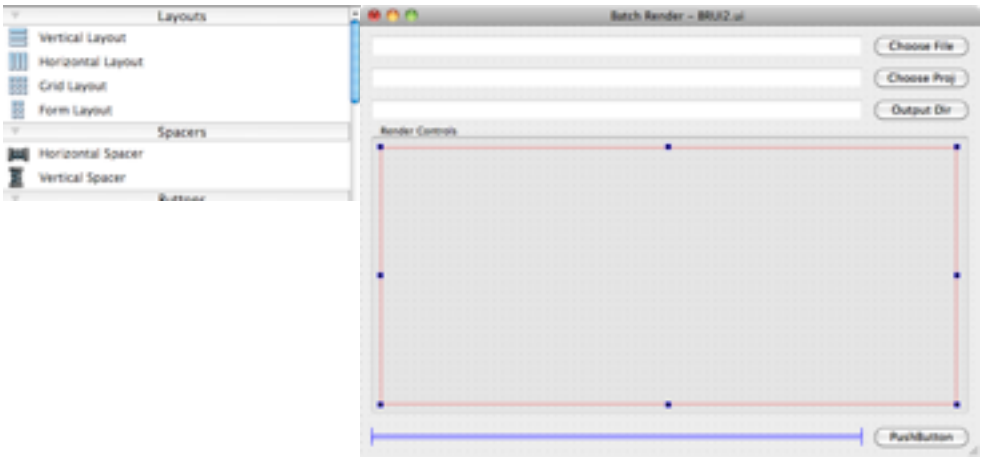


Next we add another button which will need to be spaced to fit into the correct size
First add the button and name it m_batchRender as shown

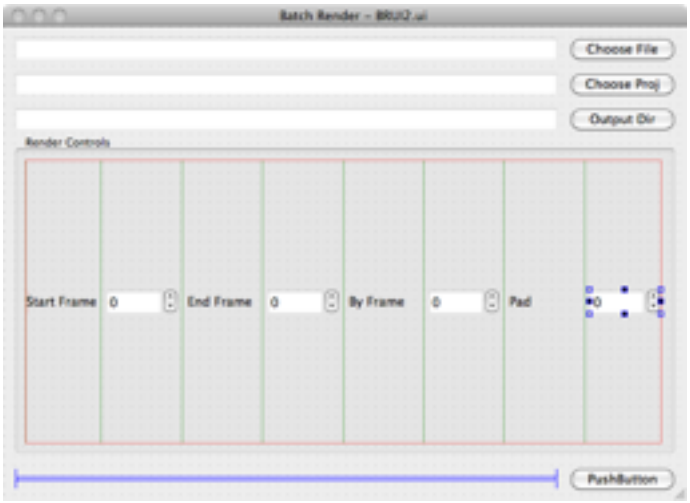


Then add a horizontal spacer to make the button fit in the correct area (you may have to add the spacer above then move the button into place)

We are now going to add the rest of the controls into the group box, we need to first add a layout to the group box, this is done by choosing the Grid Layout as shown here and scaling it to fit the group box

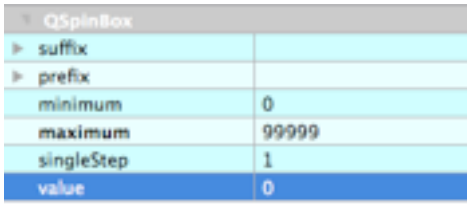


Now add the following labels and spin boxes



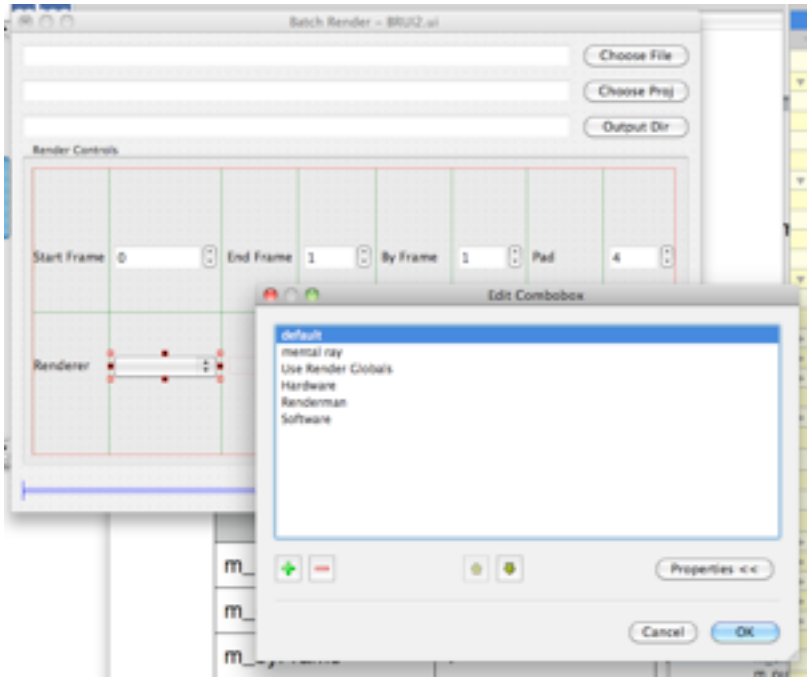
The spin boxes from left to right are called m_startFrame, m_endFrame, m_byFrame and m_pad.

We need to set some default values and ranges for each as shown



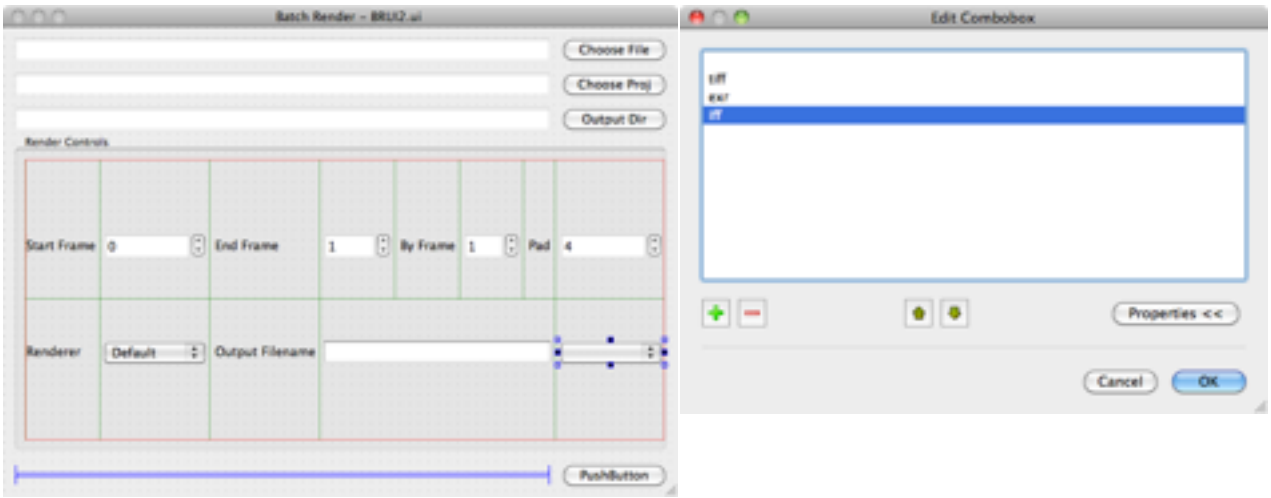
Name	minimum	maximum	value
m_startFrame	0	99999	0
m_endFrame	1	99999	1
m_byFrame	1	500	1
m_pad	0	10	4

We are now going to add a second row to the group box first a label and a combo box which we will call m_renderer as shown



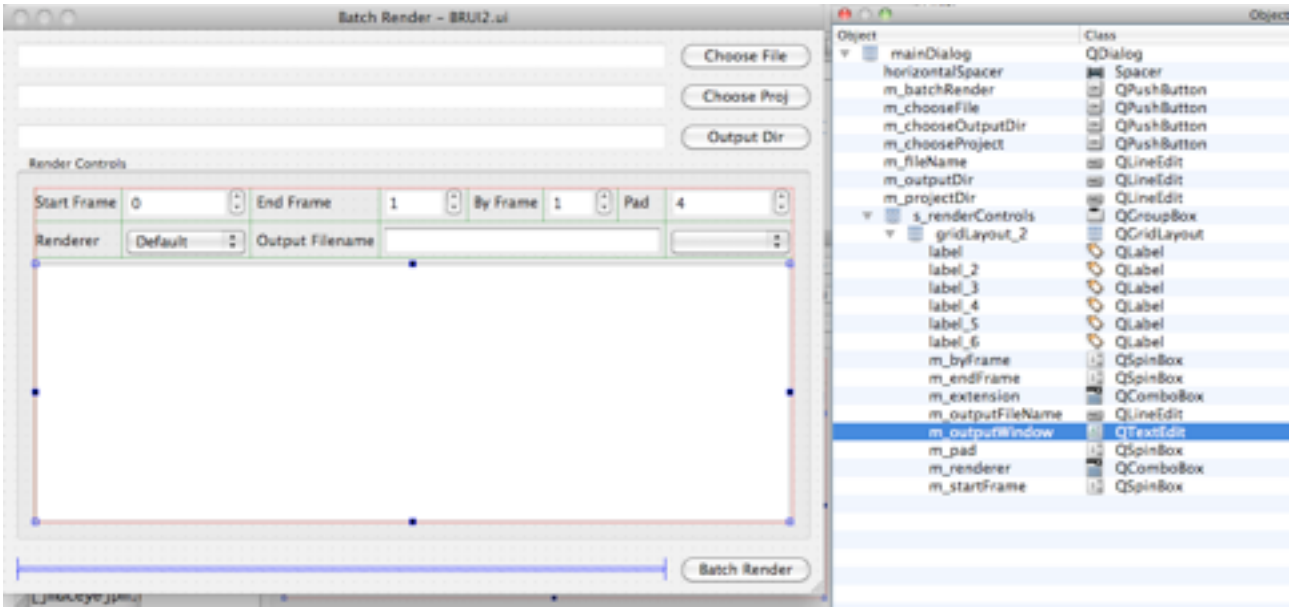
By double clicking on the combo box we can get the edit dialog and using the + button add the following text values for the different renderers.

Next we will add a text edit called m_outputFileName and a combo box called m_extension and complete the row as shown.



For the final element we are going to add a textedit so we can capture the output of the batch render, this will be called m_outputWindow and we need to set the read only flag in the property editor.

The final window should look like the following



Using PyQt

The UI file generated by QtDesigner is a simple XML file containing the layouts of the different elements. We can convert this into source code using one of the UI compilers, in this case we are developing a python application so we will use the pyuic4 compiler using the following command line.

```
pyuic4 BatchRenderUI.ui -o BatchRender.py
```

This will produce a python file for the UI elements which we will use within our own class to then create the program.

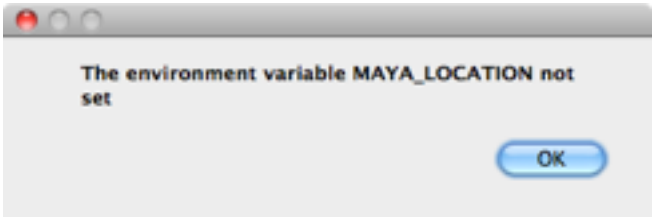
Basic Program Operation

The way the program will operate is to check that MAYA_LOCATION is in the current path, if it is not we need to tell the user and set this. This is so we can determine the correct location of the Render command in MAYA_LOCATION/bin. The basic python code to do

this is as follows

```
#!/usr/bin/python
from PyQt4 import QtCore, QtGui
from BatchRenderUI import Ui_mainDialog
import os,sys



if __name__ == "__main__":

  ResourcePath=os.environ.get("MAYA_LOCATION")
  app = QtGui.QApplication(sys.argv)


#see if the ResourcePath is set and quite if not
  if ResourcePath == None :
    msgBox=QtGui.QMessageBox()
    msgBox.setText("The environment variable MAYA_LOCATION not set")
    msgBox.show()
    sys.exit(app.exec_())

  else :

    print "ready for UI"
    sys.exit(app.exec_())
```

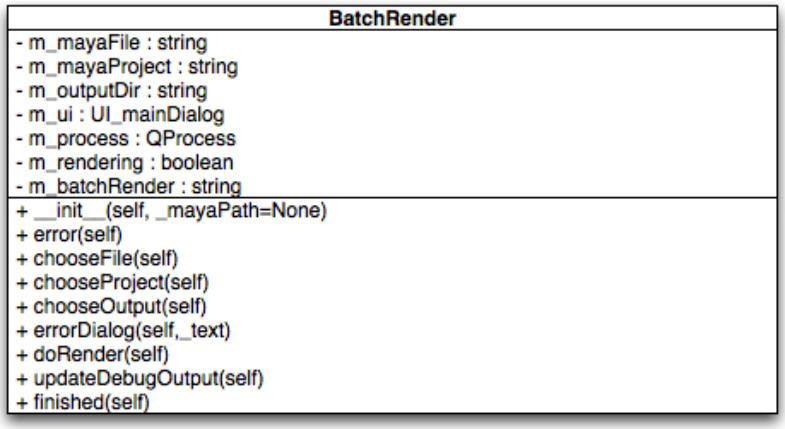
If the environment variable is not set we will get the following dialog box



To set the location we need to add export MAYA_LOCATION=:/usr/autodesk/maya2011-x64/ to our .bashrc file.

UI Class

We are now going to develop a UI class to contain the UI developed using designer and then extend it to have our own functionality and methods for the program.



The basic outline of the class init method is as follows

```
class BatchRender(Ui_mainDialog):
  def __init__(self, _mayaPath=None):

    # @brief the name of the maya file to render
    self.m_mayaFile=""
    # @brief the name of the maya project directory
    self.m_mayaProject=""
    # @brief the optional name of the output directory
    self.m_outputDir=""
    # @brief the main ui object which contains our controls
    self.m_ui=Ui_mainDialog()
    # @brief we will use this to thread our render output
    self.m_process=QtCore.QProcess()
    # @brief a flag to indicate if we are rendering or not
    self.m_rendering=False
    # @brief the batch render command constructed from the maya path
    self.m_batchRender="%sbin/Render " %(_mayaPath)
    # now we call the setup UI to populate our gui
    self.m_ui.setupUi(MainDialog)
```

This will construct the ui by calling the Ui_mainDialog constructor created from the pyuic4 command and then later call the setupUI command which is automatically generated from the pyuic compiler.

We can now update our main function to construct this object and build our dialog

```
if __name__ == "__main__":
  import sys
  app = QtGui.QApplication(sys.argv)

  ResourcePath=os.environ.get("MAYA_LOCATION")

  MainDialog = QtGui.QDialog()
  ui = BatchRender(ResourcePath)

  #see if the ResourcePath is set and quite if not
  if ResourcePath == None :
    msgBox=QtGui.QMessageBox()
    msgBox.setText("The environment variable MAYA_LOCATION not set ")
    msgBox.show()
    sys.exit(app.exec_())

  else :

    MainDialog.show()
    sys.exit(app.exec_())
```

Connecting Buttons to Methods

Qt uses the signals and slots mechanism to connect UI component actions to methods within our classes. We must explicitly connect these elements for them to work. The following code section is from the __init__ method of the BatchRender class and show this in action

```
# here we connect the controls on the UI to the methods in the class

QtCore.QObject.connect(self.m_ui.m_chooseFile, QtCore.SIGNAL("clicked()"), self.chooseFile)
QtCore.QObject.connect(self.m_ui.m_chooseProject, QtCore.SIGNAL("clicked()"), self.chooseProject)
QtCore.QObject.connect(self.m_ui.m_chooseOutputDir, QtCore.SIGNAL("clicked()"), self.chooseOutput)
QtCore.QObject.connect(self.m_ui.m_batchRender, QtCore.SIGNAL("clicked()"), self.doRender)
QtCore.QObject.connect(self.m_process, QtCore.SIGNAL("readyReadStandardOutput()"), self.updateDebugOutput)
QtCore.QObject.connect(self.m_process, QtCore.SIGNAL("readyReadStandardError()"), self.updateDebugOutput)
QtCore.QObject.connect(self.m_process, QtCore.SIGNAL("started()"), self.updateDebugOutput)
QtCore.QObject.connect(self.m_process, QtCore.SIGNAL("error()"), self.error)
QtCore.QObject.connect(self.m_process, QtCore.SIGNAL("finished()"), self.finished)
```

The m_process attribute has a number of signals to indicate the state of the process being run, this will be outlined later.

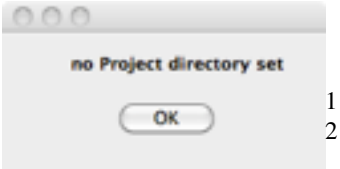
The Render process

For the batch render to run we must have a minimum of a filename and project directory set. We can check these value by seeing if the textEdit fields for each of these values are empty or not.

As part of this process we will also check to see if the startFrame value is >= endFrame value by querying the two spin boxes. The basic code for this is shown below

```
""" first we are going to check that we have the correct settings """
if self.m_mayaFile =="" :
  self.errorDialog("no␣maya␣file␣set")
  return
if self.m_mayaProject=="" :
  self.errorDialog("no␣Project␣directory␣set")
  return
if self.m_ui.m_startFrame.value() >= self.m_ui.m_endFrame.value() :
  self.errorDialog("start␣Frame␣<=␣end␣Frame")
  return
else :
  print "Doing␣render"
```

If these fail we pop up a generic dialog error box using the following code




```
def errorDialog(self,_text) :
  QtGui.QMessageBox.about(None,"Warning", _text)
```

If the criteria above are correct we can construct the Batch Render command string, this is done by building up different elements for each of the argument flags as separate strings as follows.

```
# first we need to build up the render string
renderString=self.m_batchRender
frameRange="-fnc␣name.#.ext␣-s␣%d␣-e␣%d␣-b␣%d␣-pad␣%d␣" %(self.m_ui.m_startFrame.value(),
                                                            self.m_ui.m_endFrame.value(),
                                                            self.m_ui.m_byFrame.value(),
                                                            self.m_ui.m_pad.value()
                                                            )
outputDir=""
if self.m_ui.m_outputDir.text() != "" :
  outputDir="-rd␣%s/␣" %(self.m_ui.m_outputDir.text())
outputName=""
if self.m_ui.m_outputFileName.text() !="" :
  outputName="-im␣%s␣"%(self.m_ui.m_outputFileName.text())

extension=""
if self.m_ui.m_extension.currentIndex()!=0 :
  extension="␣-of␣%s␣" %(self.m_ui.m_extension.currentText())

sceneData="-proj␣%s␣%s" %(self.m_mayaProject,self.m_mayaFile)

Renderers={0:"default",1:"mr",2:"file",3:"hw",4:"rman",5:"sw"}
rendererString="-renderer␣%s␣" %(Renderers.get(self.m_ui.m_renderer.currentIndex()))

arguments=frameRange+outputName+extension+rendererString+outputDir+sceneData;
commandString=renderString+arguments
```

The combo box for the file extensions contain the correct values for the command argument, this means that the values may be used directly using the .currentText() method of the combo box.

However the renderer string is not correct so we make a dictionary of the correct values using a integer index as the key and the string for the correct values, we then use the currentIndex value to return the integer key value and use the dictionary get() method to retrieve the correct string.

QProcess

We wish to start the batch rendering as a separate process from the rest of the system. This is so that the UI will still respond to commands whilst the batch rendering process is running, and we can also update the debug window with the text from the batch render process.

When the class is constructed we create a QProcess object called m_process, this can then be started with the command line we created above using the following code

```
self.m_process.start(commandString)
self.m_rendering = True
```

Once the process is started it will emit different signals which we can capture and respond too, we connected these signal together in the earlier code, the main one for the output of the batch render data is as follows

```
1  def updateDebugOutput(self) :
2
3    data=self.m_process.readAllStandardOutput()
4    s=QtCore.QString(data);
5    self.m_ui.m_outputWindow.append(s)
6
7    data=self.m_process.readAllStandardError()
8    s=QtCore.QString(data);
9    self.m_ui.m_outputWindow.append(s)
```

The maya batch renderer outputs most of the debug information on the stderr stream but some is also sent to the stdout stream so both streams are read to the data returned is converted to a string and added to the outputWindow.

The full code of this program can be downloaded from the following url.