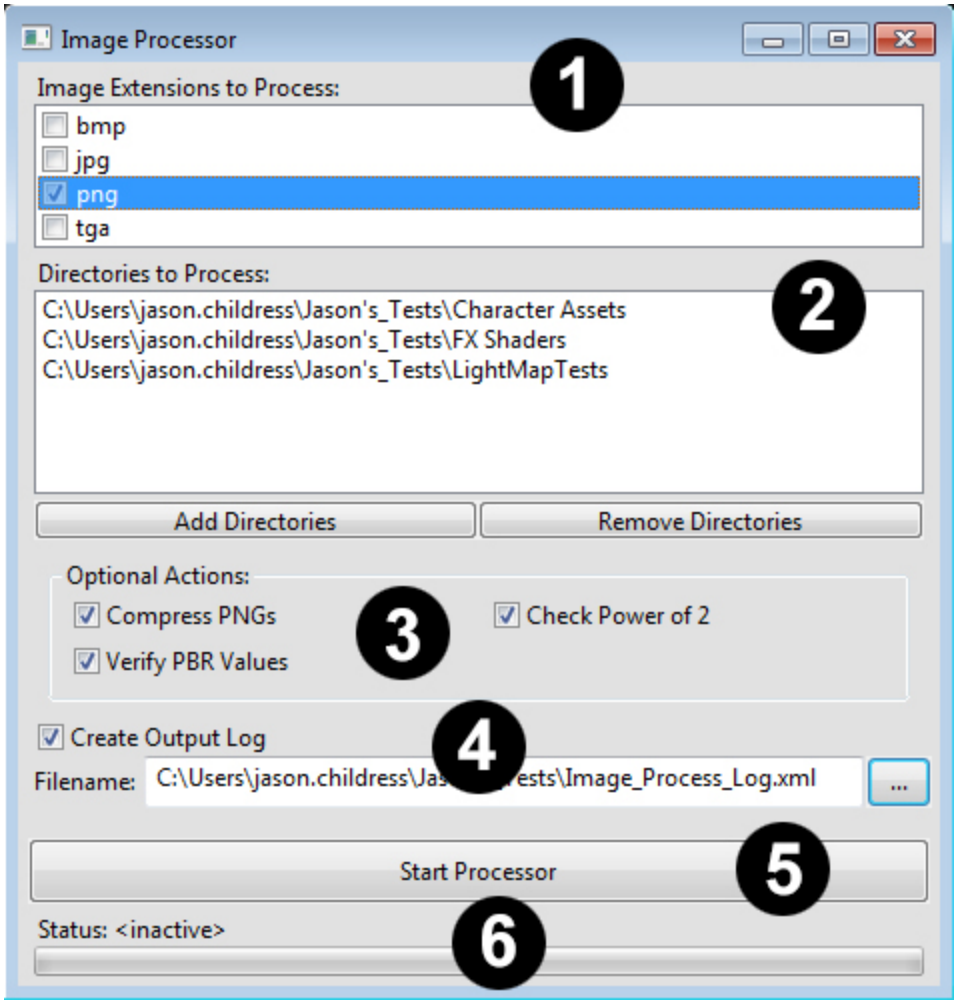
Image Processor User Guide

Using the Tool

The **Image Processor Tool** is a batch processing tool that will allow a user to select a list of directories, and perform a series of “Image Actions” on all image files within those directories. These actions can be as simple as saving a PNG file in a compressed state, verifying your image is a power of 2 or verifying images meet the proper PBR standards of your studio.

Image Actions are flexible and can be added easily by any TA or programmer. Once the new action is written, it will automatically appear in the **Optional Actions** section of the tool’s UI.

The Interface



Features

1. **Image Extensions to Process** List
    a. A list of image file types to perform the batch process on.
    b. Mix and Match any combination of extensions.
2. **Directories to Process** List
    a. The list of directories to perform the batch process on.
    b. All files within each listed directory, whose extension matches an extension from the list above, will have all of the chosen image actions performed on them.
3. **Optional Actions**
    a. All available Image Actions that can be performed on a file are listed in this box.
    b. Each action that is checked will be performed on every file, regardless of its extension.
    c. If an action is designed to work on a specific file type, and the file passed to the action is not of that type, the action will simply ignore the file.
4. **Create Output Log**
    a. When checked, will create an output log at the desired file location.
    b. The output log is in XML format.
5. **Start Processor** Button
    a. Press it to start the batch process.
6. Batch Process **Status Bar**
    a. Reports which action is currently being performed on which file.
    b. The status bar reports the percentage at which the entire batch is thru its operation.

Typical Usage

1. In the **Image Extensions to Process** list box, choose the image file types you would like to perform the batch on. You can mix and match any combination of file types.
2. Click on the **Add Directories** button to add a directory path in which all of its image files will be processed during the batch. You can add as many directories as you would like.
3. In the **Optional Actions** section, check all of the image actions you would like performed on your image files.
4. Check the **Create Output Log** option and then choose a filename which to save your output log to.
5. Press the **Start Processor** button and wait for the batch to finish. The current status of the batch will be display at the bottom of the window.

Troubleshooting

- The **Start Processor** button is disabled and I can't start the batch.
    - Ensure that all of the proper criteria is filled out before pressing the button.

        - At least one image extension must be selected.
        - At least one directory must be specified in the directories list.
        - At least one action must be checked in the actions section.
- My output log file is never created after the batch is run.
    - Make sure that you have read\write privileges on the folder in which your output log is being written to.

Reading the Log File

The log file generated by the tool will give a status report on the result of each action performed on each file. The file is organized in a way that the files that had any failures will be listed at the top of the file in the <failed> section.

XML Format

Failed Files

- Files that failed any actions are separated into their own *<failed>* section of the XML located at the top of the xml file.
- Each file has its own *<file>* element with each failed action performed on the file as an *<action>* sub-element.
- Each failed action sub-element contains the **name** of the action, and a **result** specifying why the action failed.

```
<?xml version="1.0" ?>
<root>
   <failed>
      <file filename="D:\Material Pipeline Videos\Content\Face Shadow.png">
         <action name="check_power_of_2" report="Either Width:397 or Height:391 is NOT a proper power of 2"/>
      </file>
```

All File Results

- Every single file, regardless if any of its actions failed or passed are placed in the *<complete_results>* element.
- Each file has its own *<file>* element with each action performed on the file as an *<action>* sub-element.
- Each action sub-element contains the **name** of the action, the **passed** status and a report specifying the **result** of the action performed.can

```
   <complete_results>
      <file filename="D:\Material Pipeline Videos\Content\Face Shadow.png">
         <action name="compress_png" passed="True" report="Compression saved 606.40 kbs on disk"/>
         <action name="check_power_of_2" passed="False" report="Either Width:397 or Height:391 is NOT a proper power of 2"/>
         <action name="verify_pbr_values" passed="True" report="Passed all PBR validation tests"/>
      </file>
   </complete_results>
</root>
```

Running the Tool from the Command Line

The tool can be run in its normal UI mode or in a headless mode through command line options. This makes it easy for nightly build machines or other automated verification processes to run the tool.

Command Line Arguments

- ***-headless*** - Starts the tool without any UI and automatically starts the batch process. This headless option must be accompanied by at least one directory to run the batch on, using the ***-dirs=*** argument. And at least one extension, using the ***-exts=*** argument.
- ***-dirs=*** - Specify a single or multiple list of directories to perform the batch operation on. Each directory specified must be wrapped in quotes and comma separated.
    - Example: *-dirs="C:\Game Data\Characters\Maps","C:\Game Data\Weapons\Maps"*
- ***-exts=*** - A list of image file extensions to perform the test on. Each extension specified must be wrapped in quotes and comma separated.
    - Example: *-exts="png","jpg"*
- ***-logfile=*** - The path at which to save the logfile to. If this argument is not supplied, a log will not be created.
    - Example: *-logfile="C:\Game Data\Batch Operations\Image_Process_Log.xml"*
- ***-actions=*** - A list of specific action names. Each action specified must be wrapped in quotes and comma separated.
    - The action name is the unique ***action_name*** class attribute defined by the action class in the Python code.
    - Example: *-actions="check_power_of_2","compress_png","verify_pbr_values"*

Examples of running the tool from a command line

1) Runs the tool, bringing up its normal UI and:
    a) Auto fills out the extensions with .png & .jpg.
    b) Auto fills out the directories with "C:\Game Data\Characters\Maps" and "C:\Game Data\Weapons\Maps"
    c) Auto checks the optional actions check_power_of_2 & compress_png.
    d) Auto fills out the log file location to "C:\Game Data\Batch Operations\Image_Process_Log.xml"

    The only thing the user will need to do is press the start button

```
>> python "C:\image_processor.py" -exts="png","jpg" -dirs="C:\Game
Data\Characters\Maps","C:\Game Data\Weapons\Maps"
-actions="check_power_of_2","compress_png" -logfile="C:\Game Data\Batch
Operations\Image_Process_Log.xml"
```

2) Same as example #1, except doesn't open the UI, running in headless mode instead and automatically starts the batch with the supplied settings.

```
>> python "C:\image_processor.py" -headless -exts="png","jpg" -dirs="C:\Game Data\Characters\Maps","C:\Game Data\Weapons\Maps" -actions="check_power_of_2","compress_png" -logfile="C:\Game Data\Batch Operations\Image_Process_Log.xml"
```

3) Runs the tool in headless mode, running every single possible action on the files because no ***-actions=*** argument was specified.

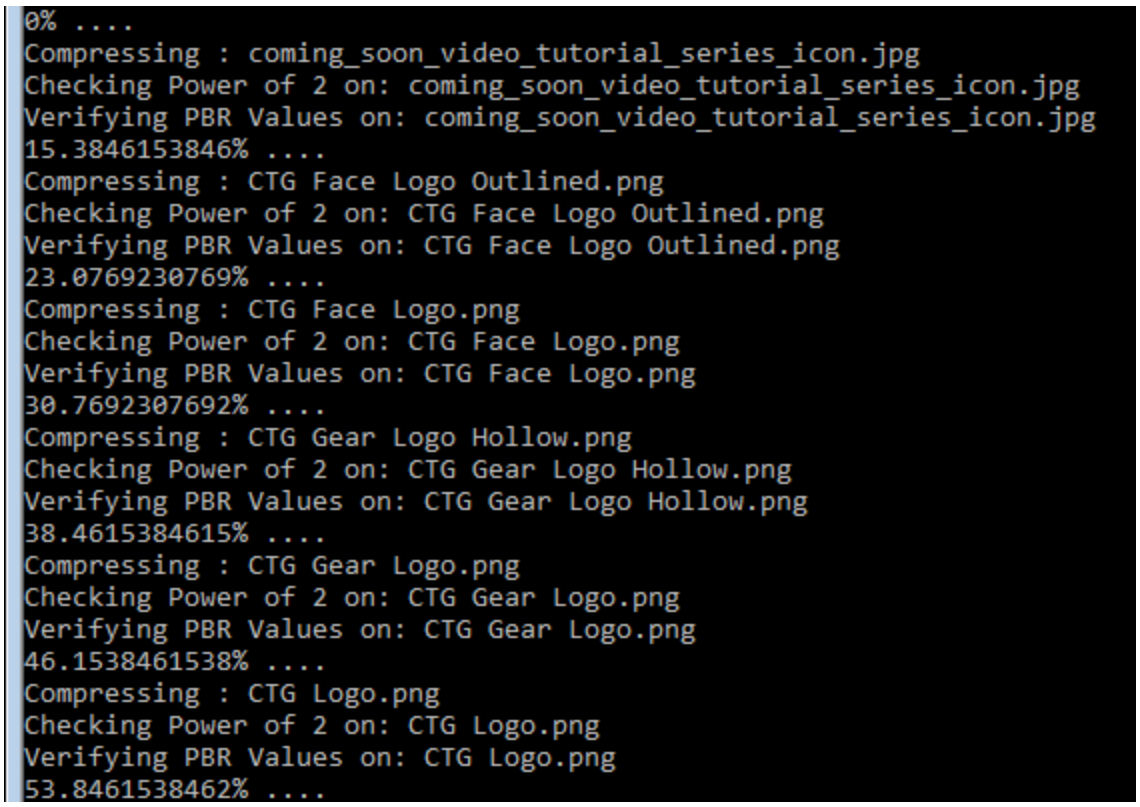
```
>> python "C:\image_processor.py" -headless -exts="png" -dirs="C:\Game Data\Characters\Maps" -logfile="C:\Game Data\Batch Operations\Image_Process_Log.xml"
```

4) Same as example #3, except no log file is generated.

```
>> python "C:\image_processor.py" -headless -exts="png" -dirs="C:\Game Data\Characters\Maps"
```

Status Updates

When the tool is running in headless mode thru the command prompt, the current progress of the batch operation is printed to the window. This is the same as if watching the status of the batch through the UI's status bar and status message.



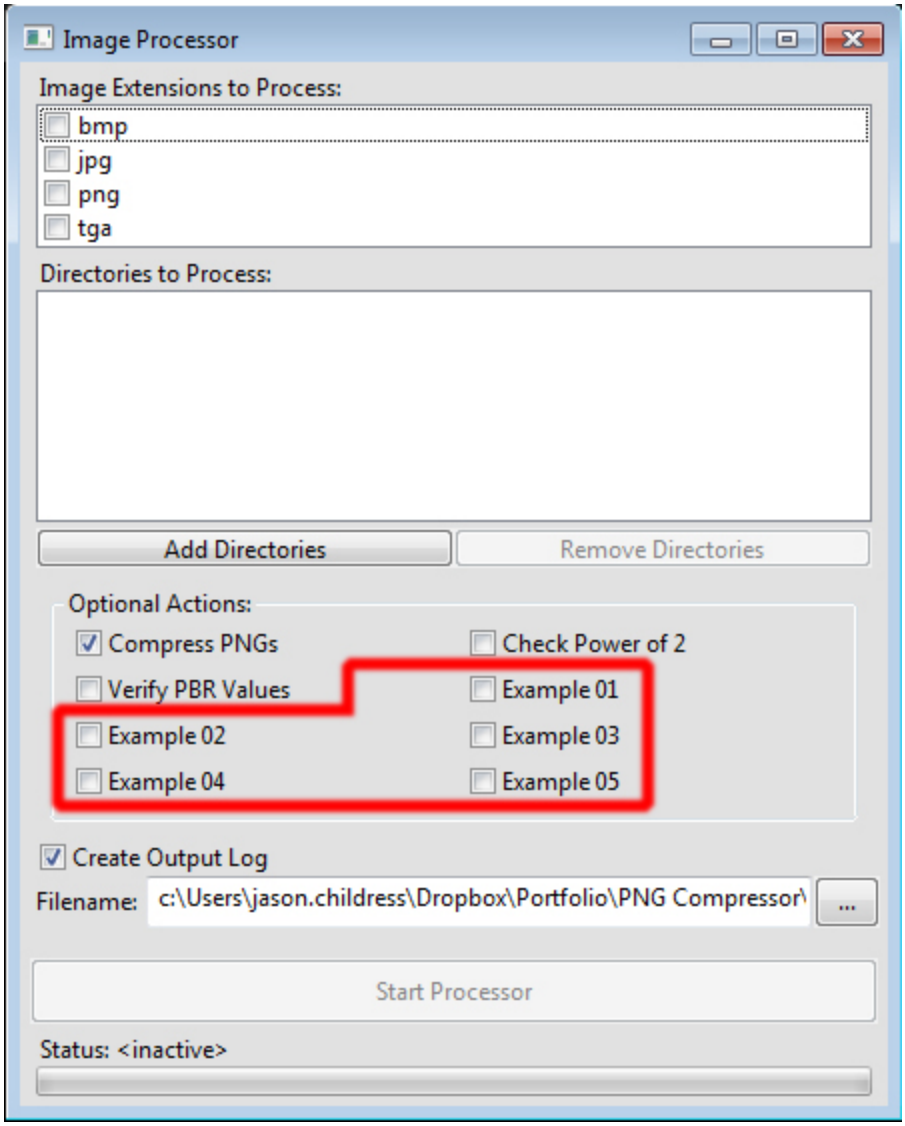
Creating New Actions

Creating new actions is easy and straightforward. It involves creating a new Python class, subclassing the ***Base_Image_Action*** class. You simply have to fill out class attributes like action_name, widget_title, status_msg and a couple others. Then write the action logic inside of the execute method and you're done.

All subclasses of ***Base_Image_Action*** will be scooped up by the UI and given a checkbox in the **Optional Actions** section of the tool.

Writing actions makes it easy to add new image checks to the tool without ever having to update any UI code.

As an example, If I edit the python file, adding 5 new dummy actions, the UI will automatically display them as such:



Anatomy of the Class

The class contains the following class attributes:

- ***action_name***: *<string>* This is the unique name of the action to help identify it in the tool. This is also the same name you would supply to the command line ***-actions=*** argument.
- ***widget_class***: *<class>* The type of widget to be displayed in the UI. NOTE: Currently, only the wx.CheckBox class is supported.
- ***widget_title***: *<string>* The human friendly label of the widget displayed in the UI.
- ***status_msg***: *<string>* The string to display in front of the image filename when the batch is performing this action. This would be the string seen in the UI's status message (At the bottom). Or in the command lines output if running the tool headless. For the Verify PBR Values action, its status_msg string would be: *"Verifying PBR Values on "*

- ***add_to_ui***: *<bool>* Determines whether this action is added to the tools UI for the user to choose.
- ***can_execute***: *<bool>* Determines if this action will be executed during the batch process.

You can make an action that is required to run and not show up in the UI by setting add_to_ui to False and can_execute to True. NOTE: If you set both to false, then the action will never run.

Overriding the Execute Method

In your new class, you must override the ***execute*** method in order for the action to do anything. The method must accept an image_obj argument which is the Image_Object class containing the image file to perform operations on.

Example of what the Compress PNGs action looks like in Python:

```python
class Action_Compress_PNG( Base_Image_Action ):
    """
    Simple action that will resave any PNG file in its compressed state
    to save memory on disk.

    Any image object handed to the execute method will be ignored unless
    it's a PNG file.
    """

    action_name     = 'compress_png'
    widget_title    = 'Compress PNGs'
    status_msg      = 'Compressing '
    add_to_ui       = True
    can_execute     = True

    @classmethod
    def execute( cls, image_obj ):
        # First check to make sure the file we're operating on is actually a PNG.
        # Otherwise, ignore this file.
        if image_obj.filename.lower( ).endswith( '.png' ):
            success     = False
            report_msg  = "Failed to complete the action for unknown reasons"

            # First, before opening the image, get it's current on disk file size
            file_stats = os.stat( image_obj.filename )
            original_file_size = file_stats.st_size

            if not image_obj.is_open:
                image_obj.open( )

            # Currently, PIL's image.save( ) method ignores the optional 'optimize' argument,
            # always saving the image in a compressed size. Now this still essentially gets
            # the job done of compressing a png on disk. But it seems like a broken implimentation
            # that I'd like to investigate in the future and fix if possible.
            image_obj.save( optimize = True )

            # After the file has been saved again, check it's file size once more to get a difference
            file_stats = os.stat( image_obj.filename )
            new_file_size = file_stats.st_size
            file_size_diff = original_file_size - new_file_size

            # If the file size has changed, report the action as passed
            if file_size_diff > 0:
                success = True
                kb_raw = str( int( float( file_size_diff ) / 1024.0 * 100 ) )
                kb_str = "{0}.{1}".format( kb_raw[ 0:-2 ], kb_raw[ -2:len( kb_raw ) ] )
                report_msg = "Compression saved {0} kbs on disk".format( kb_str )
            # Otherwise, no compression happened so the action failed
            elif file_size_diff == 0:
                success = False
                report_msg = "Compression did not save any memory on disk"
        else:
            success = False
            report_msg = "{0} is not a PNG file".format( os.path.basename( image_obj.filename ) )

        return success, report_msg
```