1 PROGRAM benchark-blender.pl

```
Blender Automated Benchmark Suite (BABS)
[cc-by] Piotr Arlukowicz, <piotao@polskikursblendera.pl>
```

Documentation: Use the Force, read the Source! OK, read at least config section below :)

USAGE

```
benchmark-blender.pl [options] -b blendfile
benchmark-blender.pl -h
```

OPTIONS

Options are specified via command line using typical getopt notation. Each option should start with - and is one letter long. Do not join options with their parameters, always add spaces around.

```
-e  file       - blender executable (default: blender)
```

The blender executable file itself. In Linux, this will be `blender`, while in windows this can be `blender.exe` or `blender.lnk`.

```
-p  path       - blender path (default: .)
```

Path to the blender. I'm usually keep it in quite a long path, and this could be a typical case. The location of: `/home/piotao/bin/blender-git/install/linux` can be expressed by several shorten terms: `%home/%git/%install`. Then, if you are using this terms, you have to put them into CONFIG section inside the script, like this:

```
%home => '/home/piotao',
%git  => 'bin/blender-git',
%install => 'install/linux',
```

The script will rebuild all config variables from those information, glueing them together. In this way you can make shortcuts and aliases, and run some additional commands along with blender testing.

```
-b  file       - blender file (HAS to be specified)
```

Normal BLEND file to be used for testing. This file will be run through blender with options given in another sections of config (or by `-c` option). Usually this has to be specified, or written into config section.

```
-c  string     - command to run blender with options, default renders only
                 one, first frame several times (defaults to "-f 1")
```

This is the command the script executes. The whole long command which should run blender rendering, passing all options to it and produce results in the form of a log, in which there is Time: (xx:yy.zz) string. Then the script can catch this results for each rendered frame and calculate results.

This command is by default composed from another config parts, for example:

```
blenderCommand => '%blenderPath/%blenderExe %options',
```

The script is building the right command from this %-something pieces. Each of them have to be specified in config sections, and you can have your own defs and strings defined. All tokens with `%` sign are replaced by theirs values defined in another parts of the config, so finally you can get the whole command. If there is a recursion problem, script will catch this and stop. So, if you define `%blenderPath` and `%blenderExe` like this:

```
blenderPath => '/bin',
blenderExe  => 'blender_x86',
```

Then, if you have your options like this:

```
options  => '%options2',
options1 => '-noaudio -noglsl -nojoystick ',
options2 => '%options1 -b %blenderFile %thr -f 1 2>&1 ',
```

Then this will give you the whole options like build as:

```
options=>'-noaudio -noglsl -nojoystick -b %blenderFile %thr -f1 2>&1 '
```

Which will then have the `%blenderFile` and `%thr` token replaced by their corresponding values. For blender file you have to specify `-b` option, and for threads number to use you have to put `-t` option.

```
-a  float      - maximal allowed change of averaged time calculated after
          each render run
```

This is the number, which stands for precision of testing. Each new test which was carried out by running blender is measured in terms of time. This time is different each time, so it is averaged, to gain the best approximation. But, how you can tell the average time is good enough? How many runs you really have to do, in order to get good average time? Please notice, that running blender three, or ten times, and averaging all times is not enough, if your times are very different from each other. So, this value can force program to run tests ENOUGH times, and after each time the new average is calculated from all gained tests. This new average is compared to the old one, and IF the old one differs no more than this value, then tests are finished. To understand this, consider this example:

Here, we have few times measured during some blender runs. They have diffrent values. How many of them is enough to have a good average value which do not change much with next tests? Let's assume we can accept 0.1 variance between two consecutive averaged results.

```
      No Time measurements         Average of times so far        Diffrence of ave
1   10                  10 (of course!)          (nothing yet)
2   11                  10.50                    0.50
3   10                  10.33                    0.16
4   12                  10.75                    0.42
5   10                  10.60                    0.15
6   10                  10.50                    0.1   -> this is the point
7   10                  10.43                    0.07
8   10                  10.38                    0.05
9
```

As you can see, during tests, we renderd 8 times, and the results were collected. For each new result, we calculated average value of all measurements and this new average value is put into the third column. Now, we can calculate the diffrence between our new average and the previous one. This value is our epsilon - the change which tells us, how much the average is changed between tests. If this change is small enough, we can stop further testing, because our result will be no better that current one! So, in this example we know, that there is only 6 run sufficient instead of 8, which can save some time. In fact, this procedure is executed as long as the change of averages is smaller than the epsilon. In our case we assumed that we are acceptig 0.1 value, so change smaller that that do not count much. This value is set by default to 1% or 0.01 (you have to specify a float number, not percent).

For lower epsilon values the computations can lasts longer and some of them can take a really long time. On my 6-cores cpu, under linux, where only 10-20 processes were run in parallel (even cron was killed tho), this procedure has taken 47 times, until the level of 0.01 was reached. For the same computer, in the save environment, the GPU has to use only 8-10 iterations.

```
      -i  int        - additional runs after time is stable for the first time
```

This is the number which tells how many additional runs the script should run AFTER the epsilon limit is reached. Look at the above example: we got the demanded minimum value in 6 iteration. Instead of stopping the calculation process, there were two additional runs performed to ensure the stability of average value. It seems that this is necessary, because there are some random fluctuations and some tests can raise the average above the limit. If that is the case, the value of repetitions is reset, and after gaining the minimum again, this number of additional runs still has to be performed.

```
      -t  int        - number of threads to use, passed to blender by -t option
```

This option is used to specify how many threads should blender use in order to render on CPU. For GPU this value is ignored. If this number is set to 0, which is the default, then all threads are used in auto-detect mode and blender will decide itself. If a value is given here, then this option appears in command line and specifies the desired number of threads to use, by applying -t NUM option to blender command line.

```
-h, --help     - prints this help and exits.
```

This is simple: it displays a short help about the program.

```
-v             - runs the program with verbose output and some debug
```

This option helped me to design this program and debug it. You do not have to use it, and I leaved it here just for those, who are curious :)

```
-l  txtfile    - blender log file to analyze only and produce results
```

If you do not want to run blender with any blender file, but you have a log output from some runs, you can just analyze that log file with no blender at all. To generate results in the same format like those generated during a real tests, specify just a single `-l logname` option, where `logname` is just a plan text file. This text file should be the complete and exact log file right from blender. It can contain multiple frames renderings, etc. but only time and last read blend file are catched. Only one log file is processed at a time and it has a precedence over blend file and blender run.

CONFIG SECTION

In the config section you can define all configuration you need to run this script without any parameters. However, configuration have it's own limitations so do not expect miracles :) This is perl, and I made this script flexible enough to suit well my needs. So, here it is.

All config definition is build as KEY => VALUE. KEY has to be unique string. VALUE can be a number, a string, etc. It can be even another structure, but then you have to know what you are doing (learn Perl! - this is very good lang)

So, if you define something like:

```
script => 'perl',
```

notice that what is on the right HAVE to be in apostrophes. The left side is automatically cited, so no apostrophes are needed. However, if you like this:

```
big idea => 'my mind',
```

This will not work, because KEY has to be a single string, so you have to cite it this way:

```
'big idea' => 'my mind',
```

After you have some such keys and values defined, you can use them in new values. Note however that you can't use them as mixed key names. So this works:

```
newidea => 'this is %script',
```

but this don't work:

```
%script => 'something',
```

Note that newidea key will be translated to 'this is perl', because before it was written that string 'script' is defined as 'perl'. So you can make some substitutions, and they are hand if you have to keep diffrent configurations inside one script. Look at this:

```
blenderCommand => '%setup1',

setup1 => '.....',
setup2 => '.....',
```

In this case one can change only one option at the start of the script, and all is changed respectively. One can then specify new configuration option via command line, putting `-c '%setup2'` parameter, and this will use a whole new, and different definition from the script config. I think this is so handy, that should suffice my needs and maybe even yours.

Remember that some of keys in the config file are reseved and they are used in the code directly. There are:

```
blenderCommand
maxAverageChange
maxRunsToEnsure
verbose
blenderFile (this is used only to ensure that the file you specified, exists)
version
```

Also, the script will create some new config keys during its work, so they are also kind-of reserved, because even if you defined them, they can be overwritten. They are:

```
framecount    - this is the numer of tests performed so far
start         - time in seconds from EPOCH when script just started
end           - time in seconds when script is about finish
totalsum      - sum of all time measurements
average       - current average value
frames        - a list of all collected time measurements (and more)
```

All other keys are not mandatory and you can change them, use them, etc.

Recursion

One can mistakenly made a recursive definitions in config, for example:

```
one => '%two',
two => '%one',
```

This naturally can lead to infinite recursion, but I thought a bit and coded simple thing to prevent this. Script will, however, fail when such a recursive definition is found. This applays also to cases where you can build token with recursive calls which at some point can lead to situation where token should be explained by itself. I'm not sure if my security code is enough, so better try not to build recursive definitions.

Output of blender command

This script works with the assumption that blender will print a log of it's rendering process to the standard output. Script is then reading this output using normal redirection pipe. If you change this behaviour, the results can be unpredictable. So, you have to assure that `blenderCommand` will be executing a command (or commands) which are able to print to the standard output. The text which is printed doesn't really matter - in the whole stream of log there is only one string searched. This is Time: xx:yy.yy value. Blender prints this time for both internal and Cycles renderers, if you run it with console output. If you plan to include some intermediate scripts between this script and blender, you should ensure that at least this line with the text Time ... is properly outputted to the output stream.

Also, to avoid messy output, the `blenderCommand` is defined with a stderr redirection to stdout. Thanks to this you don't have to see any warnings or messy error messages printed by Blender itself. This of course hides some info so you should test your blend files and render to see if everything works.

DEFAULT CONFIG

My default config section is here (in case you've lost it):

```
maxAverageChange => '0.01',
blenderExe  => 'blender',
blenderPath => '%bin/%git/%install',
blenderFile => 'any.blend',
blenderCommand => '%blenderPath/%blenderExe %options2',
options1 => '-noaudio -noglsl -nojoystick ',
options2 => '%options1 -b %blenderFile %thr -f 1 2>&1 ',
threads => 0,
verbose => 0,
version => '0.1',
start   => time,
home    => '/home/piotao',
bin     => '%home/bin',
git     => 'blender-git',
install => 'install/linux',
```

DEFAULT OPTIONS

If you would like to enter command line to redefine all config sections, here is the complete set of options:

```
-e 'blender'                        # or 'blender.exe'
-p '/path/to/blender'               # or 'c:\path\under\windoze'
-c '%blenderPath/%blenderExe -b %blenderFile %thr -f 1 '
-b blendfile.blen                   # HAS TO BE!
-a 0.01
-i 3
-t 0
```

AUTHOR

```
Piotr Arlukowicz,
Blender Foundation Certified Trainer
<piotao@polskikursblendera.pl>
Twitter: /piotao
Blender Network: /piotr-arlukowicz
```

BUGS

If there are some bugs in this code, then sorry. Nobody is perfect :) However, if you are unable to repair this script by yourself, you can try to bother me using my email. I do not promise to help, but you can always try.

DISCLAIMER

Hereby, I do not have any responsibility and will not take it if you were pissed off, harmed, killed or injured in any way using this script. Remember, this script is a one-day project made just for fun, and for testing blender on a new computer, so it is not intended to be bulletproof. It is not a rocket-science either. So, try to not shoot yourself in the foot, please :)