JSON Job Scripting

Synopsis

Tractor Job files are often represented as special purpose TCL scripts using additional job definition operators recognized by the Tractor job parser. This format was first introduced for the Alfred system, and Tractor remains compatible with nearly all Alfred constructs.

Tractor also supports an evolving set of JSON job specification formats, described here.

Introduction

Job scripts describe the work to be dispatched and monitored by Tractor. They are typically created by plug-ins in content-creation applications as they spool new rendering jobs. Thus the following syntax discussion will probably be useful only to people developing Tractor-compliant applications.

However, there is a very **simple way to generate a basic Tractor job** from the command-line or within a shell script. The tractor-spool command-line tool is used to send job scripts into the job queue managed by tractor-engine. You can also cause tractor-spool to generate a simple job script directly:

```
tractor-spool -c prman /home/yoda/new_idea.rib
```

Any text after the -c is wrapped in a job definition and will be sent to an available tractor-blade server on the farm. There are a variety of additional command-line options that can be used to modify the specifics of the job.

Job scripts support a simple hierarchical structuring scheme that can encode dependencies that, in turn, determine the order of task execution.

A JSON Job Example

Here is a simple script that defines an example task hierarchy. Assume we have two shadow maps to compute, and a final frame that references them; in this case the shadow maps must be completed before rendering can begin on the final frame.

```
{  "TractorJob": "2.0",
   "data": {
      "title": "a very short film",
      "priority": 1.0,
      "spoolcwd": "/private/tmp",
      "spooldate": "2014-03-11 12:06:55",
      "spoolfile": "x.job",
      "spoolhost": "h2o",
      "spooltime": 1394564815,
      "tags": "",
      "metadata": "",
      "comment": "",
      "envkey": [],
      "projects": ["theFilm"],
   },
   "children": [
      {
         "data": {
            "title": "Frame One",
            "commands": [
               {
                 "argv": ["render", "beauty.1.rib"],
                 "service": "PixarRender"
                 "type": "RC",
               }
            ]
         }
         "children": [
            {
               "data": {
                  "title": "Shadow A",
                  "commands": [
                     {
                       "argv": ["render", "light.1.rib"],
                       "service": "PixarRender"
                       "type": "RC",
                     }
                  ]
               }
            },
            {
               "data": {
                  "title": "Shadow B",
                  "commands": [
                     {
                       "argv": ["render", "light.2.rib"],
                       "service": "PixarRender"
                       "type": "RC",
                     }
                  ]
               }
            }

         ]

      }
   ]
}
```

This job script describes a two-level execution tree in which the task named *Scene One* will launch its command (render frame.1.rib) after the two tasks upon which it depends have completed successfully (the *Shadow*s). The nested task descriptions define the (depth-first) order of execution; hence, the render command for light.1.rib is launched first, then the one for light.2.rib follows. Then, when they are both complete, frame.1.rib is rendered. Commands are launched as separate sub-processes, in the order that they appear in the script.

Note that the two shadow commands are not dependent on each other, i.e. they aren't nested with respect to each other; hence, they represent a parallel processing opportunity. Given sufficient resources, Tractor will typically launch both commands and allow them to execute concurrently. Hence, the script defines a *launching order*, some commands may still be running when another is launched, and these concurrent commands may complete in any order. In no case, however, will a Task launch its own commands until *all* of its sub-tasks have completed their commands successfully.

The assumptions made in this example are:

- that there is an executable named *render* somewhere in the current search path.
- that the two shadow map images are created on disk where frame.1.rib expects to find them (i.e. file system references within those the RIB files are consistent).

Tractor will *not* attempt to confirm these sorts of assumptions, that is the domain of the job generator. If an error does occur, the task becomes blocked, the corresponding UI widget converts to the error state, and all tasks that depend upon the broken one will remain unexecuted (others that have no dependency on it may proceed however).

Things to note about Tasks:

- the title string will appear in the Dashboard, so it's handy to choose a brief but descriptive title.

- child tasks are listed in the -subtasks field; if there are no dependencies it can be empty or missing.
- the RemoteCmd list in the -cmds field describes the actual actions to take when the dependencies have been met; it can be empty or missing; it can also contain multiple (newline separated) RemoteCmds, which will be processed in sequence.