revl Documentation

Release 0.2.0

Christopher Crouzet

May 09, 2017

Contents

Welcome! If you are just getting started, a recommended first read is the *Overview* as it shortly covers the *why*, *what*, and *how*'s of this library. From there, the *Installation* then the *Tutorial* sections should get you up to speed with the basics required to use it.

Looking how to use a specific function, class, or method? The whole public interface is described in the *API Reference* section.

Please report bugs and suggestions on GitHub.

CHAPTER 1

User's Guide

Overview

Upon writing a piece of code for Maya, it might be interesting to know how it performs **under different conditions**, such as within scenes that are large or small, that define a deep DAG hiearchy or a flat one, that use many node types or only a few, and so on.

Following sets of user-provided commands, Revl can **pseudo-randomly generate Maya scenes** with different properties against which the behaviour of a piece of code can be observed.

The random nature of the process can also help revealing potential bugs by exposing edge cases that were not thought of, thus making it also a good tool for **unit testing**. See Wikipedia's Fuzzing page.

Note that Revl does *not* provide any sort of profiling tool to measure performances. The built-in `timeit` module as well as other open-source packages can be used for this purpose.

Features

- generates scenes by running commands a given total number of times.
- fine control over the probability distribution for each command.
- scene generations are reproducible using a fixed seed.
- extensible with custom commands.
- allows for fuzz testing.
- fast (using Maya's API, not the command layer).

Usage

```
>>> import revl
>>> commands = [
...     (4.0, revl.createTransform,),
...     (1.0, revl.createPrimitive, (), {'parent': True}),
... ]
>>> count = 100
>>> revl.run(commands, count, seed=1.23)
```

In this example, Revl invokes a total of 100 evaluations inequally shared between the two distinct commands provided, leading to create approximatively 80% of transforms, and 20% of primitives (plus their associated transforms). Also, the primitive type is picked randomly, and each primitive's transform is randomly parented under another transform from the scene, possibly creating a scene with a deep DAG hierarchy.

See also:

The *Tutorial* section for more detailed examples and explanations on how to use Revl.

Installation

Revl requires to be run from within an Autodesk Maya's Python environment. This is usually done either by running the code from within an interactive session of Maya, or through using the `mayapy` shell. A Python interpreter is already distributed with Maya so there is no need to install one.

Installing pip

The recommended[1] approach for installing a Python package such as Revl is to use `pip`, a package manager for projects written in Python. If `pip` is not already installed on your system, you can do so by following these steps:

1. Download `get-pip.py`.
2. Run `python get-pip.py` in a shell.

Note: The installation commands described in this page might require `sudo` privileges to run successfully.

System-Wide Installation

Installing globally the most recent version of Revl can be done with `pip`:

```
$ pip install revl
```

Or using `easy_install` (provided with `setuptools`):

```
$ easy_install revl
```

Development Version

To stay cutting edge with the latest development progresses, it is possible to directly retrieve the source from the repository with the help of Git:

[1] See the Python Packaging User Guide

```
$ git clone https://github.com/christophercrouzet/revl.git
$ cd revl
$ pip install --editable .[dev]
```

Note: The `[dev]` part installs additional dependencies required to assist development on Revl.

Tutorial

Revl is all about *generating pseudo-random Maya scenes* by evaluating a set of weighted commands.

The built-in commands are fairly basic and might not cover all your needs, in which case you are encouraged to *provide your own*.

Generating a Scene

A scene can be generated in two steps: creating a set of weighted commands, and evaluating it.

Each weighted command can be defined either by using the class *`Command`* or by using a tuple that follows the same structure than the *`Command`* class. Indeed, all the command definitions below are equal:

```
>>> import revl
>>> command1 = revl.Command(weight=1.0, function=revl.createTransform)
>>> command2 = revl.Command(1.0, revl.createTransform)
>>> command3 = (1.0, revl.createTransform)
```

The *command functions* being evaluated at a later stage, its arguments also need to be passed to the command definition:

```
>>> import revl
>>> command1 = revl.Command(weight=1.0, function=revl.createTransform,
...                         args=(), kwargs={'parent': True})
>>> command2 = revl.Command(1.0, revl.createTransform, (), {'parent': True})
>>> command3 = (1.0, revl.createTransform, (), {'parent': True})
```

From there, actually generating a scene is only a matter of adding the commands in a list and calling the *`run()`* function:

```
>>> import revl
>>> commands = [
...     (1.0, revl.createTransform),
... ]
>>> revl.run(commands, 100)
```

This example creates a scene with 100 transform nodes parented to the world. Setting the *`createTransform()`* function's parameter `parent` to `True` like before, already helps with adding a bit of randomness into the result:

```
>>> import revl
>>> commands = [
...     (1.0, revl.createTransform, (), {'parent': True}),
... ]
>>> revl.run(commands, 100)
```

The 100 transform nodes are now randomly parented under other transforms. The *Command.weight* attribute has no effect here, so let's see it in action:

```
>>> import revl
>>> Type = revl.PrimitiveType
>>> commands = [
...     (1.0, revl.createPrimitive, (), {'type': Type.POLY_CONE}),
...     (2.0, revl.createPrimitive, (), {'type': Type.POLY_CUBE}),
...     (5.0, revl.createPrimitive, (), {'type': Type.POLY_SPHERE}),
... ]
>>> revl.run(commands, 100)
```

Revl is invoking here a total of 100 evaluations inequally shared between the three distinct commands provided, leading the resulting scene to contain approximatively 12.5% of cones, 25% of cubes, and 62.5% of spheres.

Writing a Custom Command

The built-in commands don't offer much features. Instead they aim to be simple and fast, and as such are good contenders to be used as building blocks to compose more advanced commands.

For example, it is possible to extend the function *createPrimitive()* to set the resulting shapes as templates:

```
>>> import revl
>>> def createTemplatedPrimitive(context, type=None, name=None,
...                              parent=False):
...     primitive = revl.createPrimitive(context, type=type, name=name,
...                                      parent=parent)
...     for oShape in primitive.shapes:
...         shape = OpenMaya.MFnDependencyNode(oShape)
...         context.dg.newPlugValueBool(shape.findPlug('template'), True)
...     return primitive
>>> commands = [
...     (1.0, revl.createPrimitive),
...     (1.0, createTemplatedPrimitive),
... ]
>>> revl.run(commands, 100)
```

This example will generate a scene with as much normal primitives than templated ones.

Note: When creating a new transform node, append the resulting `maya.MObject` object to the *Context.transforms* list attribute. This will allow the function *pickTransform()* to have more transform nodes to pick from.

Note: For performance reasons, you are encouraged to make use of the *Context.dg* and *Context.dag* attributes. This buffers the DG and DAG operations, and applies them at the end of the *run()* function.

If a custom command needs to access certain features requiring the graphs to be up-to-date, then feel free to call their `doIt()` methods.

API Reference

The whole public interface of Revl is described here.

All of the library's content is accessible from within the only module `revl`.

Command Evaluation

Context	Evaluation context.
run	Randomly run weighted commands from a set.

---

class `revl.`**`Context`**`(`***kwargs*`)`

Evaluation context.

Each command function needs to define this context as first parameter.

dg

maya.OpenMaya.MDGModifier – DG modifier.

dag

maya.OpenMaya.MDagModifier – DAG modifier.

transforms

list of maya.OpenMaya.MObject – Transform nodes. Provides data for the *pickTransform()* function.

`revl.`**`run`**`(`*commands*, *count*, *seed=None*, *context=None*`)`

Randomly run weighted commands from a set.

Each command comes with a weight which determines the probabilities for that command to be run.

Use *validate()* to check if the input command set is well-formed.

Parameters

- **commands** (*list of revl.Command or compatible tuple*) – Set of weighted commands.
- **count** (*int*) – Total number of commands to be run. Setting a count greater than the number of weighted commands doesn't guarantee that each command will be run once. Some might be run multiple times instead.
- **seed** (*object*) – Hashable object to define the starting seed of the pseudo-random number generations. If `None`, the current system time is used. Running multiple times a same set of commands with a same fixed seed that is not `None` produces identitcal results.
- **context** (`revl.Context`) – Context to use. If `None`, a new one is created.

Returns The context after evaluating the commands.

Return type *revl.Context*

Examples

```
>>> import revl
>>> commands = [
...     (2.0, revl.createTransform,),
...     (1.0, revl.createPrimitive, (), {'parent': True})
... ]
>>> revl.run(commands, 100, seed=1.23)
```

Command Functions

createDagNode	Create a DAG node.
createDgNode	Create a DG node.
createPrimitive	Create a geometry primitive.
createTransform	Create a transform node.
unparent	Unparent a random transform node.

---

revl.**createDagNode**(*context*, *type*, *parent=False*)

Create a DAG node.

To create a transform node or a geometry primitive, respectively use the functions *createTransform()* or *createPrimitive()*.

Parameters

- **context** (revl.Context) – Command context.
- **type** (*maya.OpenMaya.MTypeId or str*) – Type of the node to create, for example: 'mesh', 'parentConstraint', 'pointLight', 'renderSphere', and so on.
- **parent** (*bool*) – True to parent the new DAG node under a transform randomly picked from the scene. If True but no transform could be found in the scene, then the DAG node isn't created. If False, a new transform is always created at the world and is used as the parent for the new DAG node.

Returns The new node object or *NULL_OBJ* if the parameter 'parent' was set to True but no transform could be found.

Return type maya.OpenMaya.MObject

---

revl.**createDgNode**(*context*, *type*)

Create a DG node.

To create a DAG node, use either one of the functions *createDagNode()*, *createTransform()*, or *createPrimitive()*.

Parameters

- **context** (revl.Context) – Command context.
- **type** (*maya.OpenMaya.MTypeId or str*) – Type of the node to create, for example: 'addDoubleLinear', 'bevel', 'clamp', 'lambert', and so on.

Returns The new node object.

Return type maya.OpenMaya.MObject

---

revl.**createPrimitive**(*context*, *type=None*, *name=None*, *parent=False*, *forceTransformCreation=True*)

Create a geometry primitive.

Parameters

- **context** (revl.Context) – Command context.

- **type** (*int*) – Primitive type. Available values are enumerated in the *PrimitiveType* class. If None, a primitive type is randomly picked.
- **name** (*str*) – Base name for the new transform node. If None, no name is explicitely set.
- **parent** (*bool*) – True to parent the new transform under another transform randomly picked from the scene, if any. Otherwise it is parented under the world.
- **forceTransformCreation** (*bool*) – True to always create a new transform with the shapes as child, otherwise a new transform is created only if the parameter parent is False or if no transform could be found in the scene to parent the shapes to.

Returns The new primitive, that is its generator, its transform, and its shapes.

Return type *revl.Primitive*

---

revl.**createTransform**(*context*, *name=None*, *parent=False*)

Create a transform node.

Parameters

- **context** (revl.Context) – Command context.
- **name** (*str*) – Name of the new transform node. If None, the default name is used.
- **parent** (*bool*) – True to parent the new transform under another transform randomly picked from the scene, if any. Otherwise it is parented under the world.

Returns The new transform object.

Return type maya.OpenMaya.MObject

---

revl.**unparent**(*context*)

Unparent a random transform node.

Parameters **context** (revl.Context) – Command context.

Miscellaneous

validate	Check if the commands are well-formed.
NULL_OBJ	Constant denoting an invalid object.
Command	Weighted command.
Primitive	Primitive.
PrimitiveType	Enumerator for the primitive types.
pickTransform	Randomly pick a transform.

---

revl.**validate**(*commands*)

Check if the commands are well-formed.

Parameters **commands** (*list of revl.Command or compatible tuple*) – Commands.

Raises TypeError – Some of the commands aren't well-formed.

---

revl.**NULL_OBJ = maya.OpenMaya.MObject().kNullObj**
Constant denoting an invalid object.

---

class revl.**Command**(*weight*, *function*, *args=None*, *kwargs=None*)
Weighted command.

It is not necessary to use this class to define a command as it can be done by directly using a tuple instead. But the tuple needs to be compatible with the structure defined here.

weight
float – Probability for this command to be evaluated. The value is relative to the other commands defined in the same set.

function
function – Function to evaluate. Its first argument needs to the command context *Context*.

args
tuple or None – Additional arguments to pass to the function. None is the equivalent of setting an empty tuple.

kwargs
dict or None – Keyword arguments to pass to the function. None is the equivalent of setting an empty dictionary.

---

class revl.**Primitive**(*generator*, *transform*, *shapes*)
Primitive.

An instance of this class is returned by the *createPrimitive()* function.

generator
maya.OpenMaya.MObject – Node object generating the shapes.

transform
maya.OpenMaya.MObject – Transform object.

shapes
list of maya.OpenMaya.MObject – Shape objects.

---

class revl.**PrimitiveType**
Enumerator for the primitive types.

This is used as a parameter for the *createPrimitive()* function.

NURBS_CIRCLE

NURBS_CONE

NURBS_CUBE

NURBS_CYLINDER

NURBS_PLANE

NURBS_SPHERE

NURBS_SQUARE

NURBS_TORUS

POLY_CONE

POLY_CUBE

POLY_CYLINDER

POLY_HELIX

POLY_MISC

POLY_PIPE

POLY_PLANE

POLY_PLATONIC_SOLID

POLY_PRISM

POLY_PYRAMID

POLY_SPHERE

POLY_TORUS

---

revl.**pickTransform**(*context*)

Randomly pick a transform.

Pickable transforms are listed within the *Context.transforms* attribute.

Parameters **context** (revl.Context) – Command context.

Returns The picked transform or *NULL_OBJ* if the attribute *Context.transforms* is empty.

Return type maya.OpenMaya.MObject

CHAPTER 2

Developer's Guide

Running the Tests

After making any code change in Revl, tests need to be evaluated to ensure that the library still behaves as expected.

Note: Some of the commands below are wrapped into `make` targets for convenience, see the file `Makefile`.

unittest

The tests are written using Python's built-in `unittest` module. They are available in the `tests` directory and can be fired through the `tests/run.py` file:

```
$ mayapy tests/run.py
```

It is possible to run specific tests by passing a space-separated list of partial names to match:

```
$ mayapy tests/run.py ThisTestClass and_that_function
```

The `unittest`'s command line interface is also supported:

```
$ mayapy -m unittest discover -s tests -v
```

Finally, each test file is a **standalone** and can be directly executed.

coverage

The package `coverage` is used to help localize code snippets that could benefit from having some more testing:

```
$ mayapy -m coverage run --source revl -m unittest discover -s tests
$ coverage report
$ coverage html
```

In no way should `coverage` be a race to the 100% mark since it is *not* always meaningful to cover each single line of code. Furthermore, **having some code fully covered isn't synonym to having quality tests**. This is our responsability, as developers, to write each test properly regardless of the coverage status.

Benchmarks

A set of benchmarks are also available to keep the running performances in check. They are to be found in the `benchmarks` folder and can be run in a similar fashion to the tests through the `benchmarks/run.py` file:

```
$ mayapy benchmarks/run.py
```

Or for more specificity:

```
$ mayapy benchmarks/run.py ThisBenchClass and_that_function
```

Here again, each benchmark file is a **standalone** and can be directly executed.

Note: The command line interface `mayapy -m unittest discover` is not supported for the benchmarks.

CHAPTER 3

Additional Information

Changelog

Version numbers comply with the Sementic Versioning Specification (SemVer).

v0.2.0 (2017-01-18)

Added

- Add a `validate()` function to the public interface.
- Add a `forceTransformCreation` parameter to the `createPrimitive()` function.
- Add support for coverage.
- Add a few more tests.
- Add a few bling-bling badges to the readme.
- Add a Makefile to regroup common actions for developers.

Changed

- Improve the documentation.
- Improve the unit testing workflow.
- Allow commands to be defined as lists.
- Delete the `Context`'s equality operators.
- Improve the error messages.
- Mock the `maya` module instead of running `mayapy` to generate the doc.
- Refocus the content of the readme.

- Define the `long_description` and `extras_require` metadata to setuptools' setup.
- Update the documentation's Makefile with a simpler template.
- Rework the '.gitignore' files.
- Rename the changelog to 'CHANGELOG'!
- Make minor tweaks to the code.

Fixed

- Fix the `parent` parameter's default value from the `createDagNode()` function.

v0.1.0 (2016-12-29)

- Initial release.

Versioning

Version numbers comply with the Sementic Versioning Specification (SemVer).

In summary, version numbers are written in the form `MAJOR.MINOR.PATCH` where:

- incompatible API changes increment the MAJOR version.
- functionalities added in a backwards-compatible manner increment the MINOR version.
- backwards-compatible bug fixes increment the PATCH version.

Major version zero (0.y.z) is considered a special case denoting an initial development phase. Anything may change at any time without the MAJOR version being incremented.

License

Out There

Projects using Revl include:

- bana

Index