PyOpenGLng Documentation

Release 0.1.0

Fabrice Salvaire

May 20, 2017

Contents

Welcome to the PyOpenGLng documentation!

Note: The official Home Page of PyOpenGLng is located at http://fabricesalvaire.github.io/PyOpenGLng

If you are at http://readthedocs.org then you are reading the so called *latest documentation*.

The *latest documentation* is automatically built from the git repository after each commit.

As opposite the PyOpenGLng Home Page is built manually and is thus less prone to errors.

PyOpenGLng, proudly blessed as is, is an experimental OpenGL wrapper for Python which generates the requested OpenGL API from the OpenGL XML API Registry and use an automatic translator to map the C API to Python. The interface between C and Python is supported by ctypes and also by CFFI which paves the way to use the pypy interpreter.

By design this wrapper supports all the OpenGL version, but it focus towards the programmable pipeline and the most recent OpenGL API. On Linux desktop, Mesa 3D Graphics Library release 10 (November 2013) supports the OpenGL 3.3 API for Intel HD GPU.

The Python package provides three components:

- an Oriented Object API to the OpenGL XML registry,
- a ctypes and CFFI wrapper,
- an experimental high level API.

Warning: We should test all the API to claim a compliance with the OpenGL API. Since the OpenGL API becomes more and more large and complex over the release, such attempt would require a huge amount of work. Up to now only a part of the API was tested successfully.

CHAPTER 1

Bibliography

The followings list of links provides an overview on the topic:

- PyOpenGL - the de facto standard OpenGL Python binding
- Vispy - a high-performance interactive 2D/3D data visualization library

CHAPTER 2

Installation

The procedure to install PyOpenGLng is described in the *Installation Manual*.

CHAPTER 3

Testing

Some examples are provided with PyOpenGLng, see the *example section*.

CHAPTER 4

Documentation

CHAPTER 5

Overview

Installation

Dependencies

PyOpenGLng requires the following dependencies:

- Python 2.7
- Numpy
- freetype-py
- PyQt 4.9 for the high level API and the examples

Installation from PyPi Repository

PyOpenGLng is made available on the PyPI repository at https://pypi.python.org/pypi/PyOpenGLng

Run this command to install the last release:

```
pip install PyOpenGLng
```

Installation from Source

The PyOpenGLng source code is hosted at https://github.com/FabriceSalvaire/PyOpenGLng

To clone the Git repository, run this command in a terminal:

```
git clone git@github.com:FabriceSalvaire/PyOpenGLng.git
```

Then to build and install PyOpenGLng run these commands:

```
python setup.py build
python setup.py install
```

Examples

Some examples are provided with the PyOpenGLng source in the `examples` directory, we will presents them in the followings.

Note: PyOpenGLng was only tested on the Linux platform.

First of all, either you have to install PyOpenGLng (using virtualenv for example) or you must setup your `PYTHONPATH` environment variable. For example enter these commands in a Linux terminal:

```
cd pyopenglng
export PYTHONPATH=$PWD:$PYTHONPATH
```

The script `test-high-level-api.py` shows an usage of the high level API for both OpenGL version 3 and 4. The first one is intended for platform running the Mesa 3D Graphics Library which support the OpenGL 3.3 API on Intel HD GPU (since release 10.0 / november 2013). The second one is intended for platform running a proprietary driver like the one provided by Nvidia which implements OpenGL up to the version 4.4.

To run this example, enter the following commands in a Linux terminal:

```
cd examples
python test-high-level-api.py --opengl=v3 # designed to run on Mesa
python test-high-level-api.py --opengl=v4 # require a proprietary driver
```

Tools

The command **`query-opengl-api`** in the `bin` directory provides a tool to query the OpenGL API using the `PyOpenGLng.GlApi` module.

The online help could be printed on the terminal using the command:

```
> query-opengl-api --help

usage: query-opengl-api [-h] --api {gl,gles} --api-number API_NUMBER
                        [--profile {core,compatibility}] [--validate]
                        [--translate-type] [--build-wrapper] [--summary]
                        [--list-enums] [--list-commands]
                        [--list-multi-referenced-pointer-commands]
                        [--list-computed-size-commands]
                        [--list-multi-pointer-commands] [--enum ENUM]
                        [--command COMMAND] [--man MAN]

A tool to query the OpenGL API

optional arguments:
  -h, --help            show this help message and exit
  --api {gl,gles}       API (default: None)
  --api-number API_NUMBER
                        API number (default: None)
  --profile {core,compatibility}
                        API profile (default: core)
  --validate            validate xml file (default: False)
  --translate-type      translate gl to c type (default: False)
  --build-wrapper       Build wrapper (default: False)
  --summary             summary (default: False)
  --list-enums          list enums (default: False)
  --list-commands       list commands (default: False)
  --list-multi-referenced-pointer-commands
```

```
                        list commands having size parameter used by more than
                        one pointer parameter (default: False)
  --list-computed-size-commands
                        list commands having a computed size parameter
                        (default: False)
  --list-multi-pointer-commands
                        list commands having a multi-pointer parameter
                        (default: False)
  --enum ENUM           Show enum property (default: None)
  --command COMMAND     Show command prototype (default: None)
  --man MAN             Show man page (default: None)
```

We will presents the main functions in the followings.

In any case, you have to select an API, its number and profile. As example we will use the OpenGL V3.3 core profile API.

You can print a summary on the API using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --summary

OpenGL API gl 3.3 profile: core
  - Number of Enums:      797
  - Number of Commands:   344
```

You can list all the enumerants using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --list-enums

Enum GL_ACTIVE_ATTRIBUTES = 0x8b89
Enum GL_ACTIVE_ATTRIBUTE_MAX_LENGTH = 0x8b8a
Enum GL_ACTIVE_TEXTURE = 0x84e0
...
```

You can list all the commands using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --list-commands

glActiveTexture
glAttachShader
glBeginConditionalRender
...
```

You can print the definition of an enumerant using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --enum GL_ACTIVE_
↪ATTRIBUTES

Enum GL_ACTIVE_ATTRIBUTES = 0x8b89 (type: None, alias: None, api: None, comment:␣
↪None)
```

You can print the definition of a command using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --command␣
↪glActiveTexture

void glActiveTexture (GLenum texture)
```

And you can translate the GL type to C using:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --command␣
↪glActiveTexture --translate-type
```

```
void glActiveTexture (unsigned int texture)
```

To get the translation use:

```
> query-opengl-api --api=gl --api-number=3.3 --profile=core --build-wrapper --
↪command glDeleteBuffers

glDeleteBuffers - delete named buffer objects

glDeleteBuffers (InputArrayWrapper<const unsigned int * [n]> buffers)

void glDeleteBuffers (GLsizei n, const GLuint * [n] buffers)
```

API Documentation

This is the auto-generated API documentation for the PyOpenGLng library.

Warning: The API documentation is automatically generated from the docstrings in the source using the Sphinx tool. This way to produce the documentation is know to be perfectible actually, but not too bad.

Contents:

Prototype Translation

The C language defines strictly by a prototype how to use the parameters and the output of a function.

The OpenGL API only use fundamental types, pointer and array for parameters and return. The API do not use structures or unions which are compound types. Example of fundamental types are integer, float and char. Pointer or array are used to pass or return a multiple of a fundamental types. Also the API use pointer parameters to return more than one items, as usual in C since a function can only return one item at once. Pointers are also used to write data at a given place, which act as a kind of input-output parameter <<???>>. We know if a pointer parameter is an input by the presence of the *const* qualifier.

C Arrays do not embed their sizes, thus this information must be provided either implicitly, either explicitly or using a sentinel, as for null-terminated string. The XML schema that defines the OpenGL API provides partially this information. We can know exactly the size of an array, if the array size is fixed or passed by a second parameter. However it the size depends of the context, for example a query enum, the size is tagged as a computation from one or more parameters. But for these cases the schema do not provide a formulae described by a meta-language.

The main concept of this wrapper is to use the XML data to generate a Python API with a natural behaviour.

Fundamental types

Prototypes which are only made of fundamental types, for example:

```
void glBindBuffer (GLenum target, GLuint buffer)
->
glBindBuffer (ParameterWrapper<unsigned int> target, ParameterWrapper<unsigned int>
↪ buffer)
```

are translated to `ParameterWrapper` and passed by copy. These parameters are the input of the function.

Input parameters passed as pointer

Input parameters passed as pointer are necessarily qualified as const.

They are managed by an `InputArrayWrapper` when the size is not tagged as computed in the XML registry. For these cases, the pointer parameter takes the place of the size parameter and its parameter slot is removed of the prototype in Python. The size is automatically filled by the wrapper. For example:

```
void glBufferData (GLenum target, GLsizeiptr size, const void * [size] data,␣
↪GLenum usage)
->
glBufferData (ParameterWrapper<unsigned int> target,
              InputArrayWrapper<const void * [size]> data,
              ParameterWrapper<unsigned int> usage)

void glUniform3fv (GLint location, GLsizei count, const GLfloat * [count] value)
->
glUniform3fv (ParameterWrapper<int> location, InputArrayWrapper<const float *␣
↪[count]> value)
```

<<INVERSE Pointer <-> Size>> <<case with more than one pointer>>

The array can be passed as an iterable or a numpy array. A missing information in the actual schema is due to the fact the size can represents the number of elements or a number of byte. Usually a generic pointer indicates a size specified in byte. <<TO BE CHECKED>>

Some functions have ** pointer parameters. The function *glShaderSource* is an interesting case since it features this kind of pointer and the size parameter is used for two parameters:

```
void glShaderSource (GLuint shader, GLsizei count, const GLchar ** [count] string,␣
↪const GLint * [count] length)
->
glShaderSource (ParameterWrapper<unsigned int> shader,
                InputArrayWrapper<const char ** [count]> string)
```

According to the specification, the *string* parameter is an array of (optionally) null-terminated strings and the *length* pointer must be set to NULL in this case.

Output parameters passed as pointer

These parameters are not qualified as const and are managed by an `OutputArrayWrapper` when the size is not tagged as computed.

The pointer parameter takes the place of the size parameter and its parameter slot is removed of the prototype in Python. The size is automatically filled by the wrapper.

If the pointer is generic, then the array is passed as an Numpy array and the size is specified in byte. For example:

```
void glGetBufferSubData (GLenum target, GLintptr offset, GLsizeiptr size, void *␣
↪[size] data)
->
glGetBufferSubData (ParameterWrapper<unsigned int> target, ParameterWrapper
↪<ptrdiff_t> offset,
                    OutputArrayWrapper<void * [size]> data)
-> None
```

If the pointer is of *char type, then the size is passed by the user and a string is returned. For example:

```
void glGetShaderSource (GLuint shader, GLsizei bufSize, GLsizei * [1] length,␣
↪GLchar * [bufSize] source)
->
```

```
glGetShaderSource (ParameterWrapper<unsigned int> shader, OutputArrayWrapper<char␣
↪* [bufSize]> source)
-> source
```

If the user passes an Numpy array, then the data type is checked and the size is set by the wrapper. If the user passes a size, then a Numpy array (or a list) is created and then returned:

```
void glGenBuffers (GLsizei n, GLuint * [n] buffers)
->
glGenBuffers (OutputArrayWrapper<unsigned int * [n]> buffers)
```

Parameter passed by reference

A parameter passed by reference is identified in the prototype as a non const pointer with a fixed size of 1. Reference parameter are removed in the Python prototype and their values set by the command are returned in their prototype order. For example, this function features 3 parameters passed by reference:

```
void glGetActiveUniform (unsigned int program, unsigned int index, int bufSize,
                         int * [1] length, int * [1] size, unsigned int * [1] type,
                         char * [bufSize] name)
->
glGetActiveUniform (ParameterWrapper<unsigned int> program, ParameterWrapper
↪<unsigned int> index,
                    OutputArrayWrapper<char * [bufSize]> name)
-> name, length, size, type
```

Parameter passed as pointer

When the size is tagged as computed, parameters are managed by a `PointerWrapper` and all the parameters involved in the the size determination must be passed as input parameter:

```
void glBindAttribLocation (GLuint program, GLuint index, const GLchar * name)
->
glBindAttribLocation (ParameterWrapper<unsigned int> program, ParameterWrapper
↪<unsigned int> index,
                      PointerWrapper<const char *> name)
```

<<Fixme null-terminated>>

For example this function features a generic pointer *pixels* which must be passed as an Numpy array:

```
void glTexImage1D (GLenum target, GLint level, GLint internalformat, GLsizei width,
↪ GLint border,
                   GLenum format, GLenum type, const void * [COMPSIZE(format,type,
↪width)] pixels)
->
glTexImage1D (ParameterWrapper<unsigned int> target, ParameterWrapper<int> level,
              ParameterWrapper<int> internalformat, ParameterWrapper<int> width,
              ParameterWrapper<int> border, ParameterWrapper<unsigned int> format,
              ParameterWrapper<unsigned int> type,
              PointerWrapper<const void * [COMPSIZE(format,type,width)]> pixels)
-> None
```

This example is interesting, since the *width* parameter can be deduced from the shape of the Numpy array.

This function features a typed pointer:

```
void glGetIntegerv (GLenum pname, GLint * [COMPSIZE(pname)] data)
->
```

```
glGetIntegerv (ParameterWrapper<unsigned int> pname, PointerWrapper<int *␣
↪[COMPSIZE(pname)]> data)
-> None
```

Return parameter passed as pointer

The wrapper only supports null-terminated string, for example:

```
const GLubyte * glGetString (GLenum name)
->
glGetString (ParameterWrapper<unsigned int> name)
```

Indexes

- genindex
- modindex
- search