pyOCCT Documentation

Release 7.4.0.0

Trevor Laughlin

Nov 25, 2021

Contents

CHAPTER 1

About

The **pyOCCT** project provides Python bindings to the OpenCASCADE geometry kernel via pybind11. Together, this technology stack enables rapid CAD/CAE/CAM application development in the popular Python programming language.

## Enabling technology The pyOCCT core technology stack includes:

- OpenCASCADE: Open CASCADE Technology (OCCT) is an object-oriented C++ class library designed for rapid production of sophisticated domain-specific CAD/CAM/CAE applications.
- pybind11: A lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code.

CHAPTER 2

Design Considerations

This section describes some fundamental assumptions and implementation details for the pyOCCT project, most of which relate to pybdind11 usage. Feedback is welcomed on these topics based on usage experience and expertise.

2.1 Organization

The pyOCCT project, source code, and binaries are generally organized by OpenCASCADE packages (over three hundred so far), which eventually get organized into separate Python modules. Each pyOCCT source file represents an OpenCASCADE package and gets compiled into a Python module and contains all the entities of that package. Import statements and call guards in the pyOCCT source are used as needed. This results in a general structure of:

```
from OCCT.PackageName import ClassName
```

to be used in Python when importing the various pyOCCT modules. One major advantage of this approach has been the ability to modify and compile each module separately rather than a single large binary, reducing build time and improving maintainability.

2.2 Static Methods

To avoid naming conflicts with regular instance methods, the names of static methods are modified by appending a trailing underscore. For example, the code

```
edge = TopoDS.Edge_(shape)
```

uses the static method `TopoDS::Edge` to downcast a shape to an edge if possible. The trailing underscore was selected as a global rule to avoid naming conflicts and inform the user they are calling a static method.

2.3 Templates

The OpenCASCADE library is a large and complex codebase that makes use of modern C++ features including templates. Compared to earlier versions, the latest releases of OpenCASCADE define a number of different types by instantiating a set of core class templates. To more closely follow the OpenCASCADE architecture and avoid repetitive code, function templates are used whenever possible to bind types derived from an OpenCASCADE class template. In the appropriate module, the function template can be called with essentially the same parameters as its OpenCASCADE counterpart, with the addition of the current pybind11 module and its desired name.

As an example, binding the type `TopoDS_ListOfShape` in the `TopoDS` module source code is:

```
bind_NCollection_List<TopoDS_Shape>(mod, "TopoDS_ListOfShape");
```

where `mod` is the pybind11 module. In the `NCollection` template header file included in the `TopoDS` module source code, the function template looks similar to:

```
template <typename TheItemType>
void bind_NCollection_List(py::object &mod, std::string const &name) {

    py::class_<NCollection_List<TheItemType>, NCollection_BaseList> cls(mod, name.c_
↪str());
    cls.def(py::init<>());
    // continued in source...

};
```

This seems to be an efficient and maintainable implementation, but feedback and suggestions are welcomed.

2.4 Smart Pointers

The first and most critical decision was what smart pointer to use for binding Python classes. Both `std::unique_ptr` and `std::shared_ptr` are supported out of the box by pybind11, with `std::unique_ptr` being used by default.

OpenCASCADE also implements its own custom smart pointer referred to as a handle in the library. This `opencascade::handle<T>` class template is their own implementation of an intrusive smart pointer used with the `Standard_Transient` class and its descendants.

The following approach was taken for smart pointer selection:

1. Use `std::unique_ptr` for all types if not a descendant of `Standard_Transient`
2. Use `opencascade::handle` for all descendants of `Standard_Transient`

To enable the use of `opencascade::hande` the following macro is applied:

```
PYBIND11_DECLARE_HOLDER_TYPE(T, opencascade::handle<T>, true);
```

where `true` is used since it uses intrusive reference counting. So far this seems to be a workable and convenient approach, but again feedback is welcomed.

2.5 Non-public Destructors

One reason the `std::unique_ptr` was chosen as described above is the ability to handle types non-public destructors. This is described here in the pybind11 documentation. A number of OpenCASCADE types make use of non-public destructors and the pybind11 helper class `py::nodelete` is used when binding these types.

As a result of using `py::nodelete` in some types, it was found that types derived from those with non-public destructors must have some type of helper class in the `std::unique_ptr` instantiation otherwise a compile error would result. It was unclear whether this was a compiler or pybind11 issue, but the remedy at the time was to implement a "dummy" helper class as:

```
template<typename T> struct Deleter { void operator() (T *o) const { delete o; } };
```

and use in binding source like:

```
// Base type with non-public destructor
py::class_<Base, std::unique_ptr<Base, py::nodelete>>

// Derived type with public destructor
py::class_<Foo, std::unique_ptr<Foo, Deleter<Foo>>, Base>
```

This `Deleter` template pattern was applied to all types with public destructors to better support the automation of the binder generation tool. Early tests seemed to indicate that this worked as expected (i.e., instances were deleted as the Python reference count dropped to zero), but the implications of this approach may not be entirely understood and feedback is welcomed.

2.6 Iterators

Some types support iteration like `NCollection_List<TheItemType>` which is used as the template for the `TopoDS_ListOfShape` type. So now the user can do something like:

```
from OCCT.TopoDS import TopoDS_ListOfShape


shape_list = TopoDS_ListOfShape()
shape_list.Append(item1)
shape_list.Append(item2)

for item in shape_list:
    do something...
```

Enabling iterators is done by defining a `__iter__` method for the type if the type also has `begin` and `end` methods, the assumption here being that this type is an iterator. For the example above, both `NCollection_List<TheItemType>::begin` and `NCollection_List<TheItemType>::end` are present so the binder generation tool automatically implement the method:

```
cls.def("__iter__", [](const NCollection_List<TheItemType> &s) { return py::make_
↪iterator(s.begin(), s.end()); }, py::keep_alive<0, 1>());
```

This seems to be a useful approach but it dependent on function names.

2.7 Overriding Virtual Functions

The capability to override virtual functions defined in a C++ class in Python is provided by pybind11 and described here. Initial attempts to provide this functionality to pyOCCT were made using trampoline classes but proved to be difficult and complex to implement via the automated generation tool. Therefore, this capability is not provided in pyOCCT and typical usage thus far has not required it.

2.8 Reference Arguments

Passing arguments by mutable references and pointers is common in C++, but certain Python basic types (`str`, `int`, `bool`, `float`, etc.) are immutable and will not behave the same way. This is described in detail in the pybind11 docs. For example, this method passes in the `First` and `Last` arguments by reference and are floats which are modified in place while the method returns the underlying curve. In Python, providing these last two parameters will have no affect. To remedy this, some logic is built into the binding generation tool that attempts to recognize Python immutable types that are passed by reference (and without `const`) and instead return them in a tuple along with the regular return argument. To maintain overload resolution order, "dummy" parameters are still required to be input. The example in Python now becomes something like:

```
curve, first, last = BRep_Tool.Curve_(edge, 0. ,0.)
```

So far this has proven to be a reliable approach but is dependent on the logic and assumptions described above.

2.9 Exceptions

Exception handling is supported by pybind11 and described here in the pybind11 documentation. How to best handle exceptions raised by the OpenCASCADE library on the Python side has not yet been fully explored. A minimal attempt can be found at the bottom of the *Standard.cpp* source file and is also shown below.

```
// Register Standard_Failure as Python RuntimeError.
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    }
    catch (const Standard_Failure &e) {
        PyErr_SetString(PyExc_RuntimeError, e.GetMessageString());
    }
});
```

This seems to catch and report some errors in Python but not all. Alternative approaches are improvements are needed. This small implementation was placed in the `Standard` module since most, if not all, modules import this module at some level.

CHAPTER 3

Known Issues

This is a summary of some known issues:

- Methods like `ShapeAnalysis_FreeBounds::ConnectEdgesToWires` take in a `TopTools_HSequenceOfShape` which is modified on the C++ side to contain the resulting wires. In the source, they use `owires = new TopTools_HSequenceOfShape` to I think clear the list. At this point I think this breaks the associativity to the Python variable as the provided variable is not changed. For now, this is avoided by using a lambda function in the bindings and the resulting wires are returned rather than modified as an input. So far only trial and error has detected these issues and they are usually fixed on a case-by-case basis.
- While pyOCCT provides coverage for a significant amount of the OpenCASCADE codebase, there are exceptions. An error will be thrown if a needed type is not registered by pybind11. Sometimes it's just a small matter of patching the source to expose a type, function, or attribute. It could also be omitted for a reason and the user is encouraged to investigate the issue and determine the root cause. Issues (and hopefully resolutions) can be submitted using GitHub Issues and Pull Requests.
- Arrays are not supported but have not been encountered during typical usage. Resolving this mostly just requires a better understanding of how to handle arrays within pybind11.
- Support for nested classes and types is mixed. This is well supported in pybind11 but its a matter of implementation detail and complexity in the automated binding generation tool. Things can be fixed manually as needed but another pass at this is needed in the binding generator tool.

CHAPTER 4

Overview

For now, extensions are relatively small, lightweight modules intended to streamline basic OpenCASCADE functionality and make it more *pythonic*. They should be small in scope and provide relatively generic capability to enable users to more quickly develop their own applications. Development of large-scale, special purpose toolkits or applications is outside the scope of native pyOCCT functionality.

CHAPTER 5

Exchange

The `Exchange` extension provides tools for data exchange including reading and writing BREP, STEP, and/or IGES files. The tools can be imported as:

```
from OCCT.Exchange import *

shape = ExchangeBasic.read_step('model.step')
```

The following tools are available:

Table 1: OCCT.Exchange tools.

Name	Description
`ExchangeBasic`	Basic read/write static methods.

CHAPTER 6

Visualization

A minimal viewing tool is provided in the `Visualization` extension. It can be imported as:

```
from OCCT.Visualization import WxViewer

v = ViewerWx()
v.add(*args)
v.start()
```

This is intended to provide only a minimum capability to display shapes to the screen. Examine the source further for other methods and properties.

The following tools are available:

CHAPTER 7

How to Cite pyOCCT

The following Bibtex template can be used to cite the pyOCCT project in scientific publications:

```
@misc{pyOCCT,
   author = {Trevor Laughlin},
   year = {2020},
   note = {https://github.com/trelau/pyOCCT},
   title = {pyOCCT -- Python bindings for OpenCASCADE via pybind11}
}
```

CHAPTER 8

PythonOCC Comparison

The overall organization between **pyOCCT** and PythonOCC is very similar. The most noticeable difference is that the installed package is called `OCCT` instead of `OCC` and the concept of handles as described below.

8.1 Static Methods

In OCC, static methods are converted to module level methods with their name following the format `modulename_MethodName()`. In pyOCCT, static methods are within the class but have a trailing underscore. The trailing underscore was needed to avoid naming conflicts with regular class methods. For example, the method to convert a generic `TopoDS_Shape` to a `TopoDS_Edge` in PythonOCC is:

```
from OCC.TopoDS import topods_Edge
```

In pyOCCT, this is now:

```
from OCCT.TopoDS import TopoDS
```

and the method is called as:

```
edge = TopoDS.Edge_(shape)
```

8.2 GetHandle() and GetObject()

In PythonOCC, a Python object wrapping an OpenCASCADE type usually had a method called `GetHandle()` which would return a `Handle_*` instance (e.g., `Handle_Geom_Curve`), or a `GetObject()` method to return the underlying object if you have a `Handle_*` instance on the Python side. The OpenCASCADE `opencascade::handle<Type>` is their own implementation of a smart pointer for memory management. In pyOCCT, the binding technology actually uses the OpenCASCADE handle as a custom smart pointer (everything is wrapped by a smart pointer in pybind11) so on the Python side the wrapped type actually serves as **both** the object and the handle. Methods that returned a `Handle_*` instance in PythonOCC will now return the specific type (i.e., `Handle_Geom_Curve` now just comes back as a `Geom_Curve`). There is no more `GetHandle()` or `GetObject()` methods. Methods and/or classes that require a handle as an input can now just be supplied the pyOCCT instance.

8.3 Return Types

In pybind11, return types are resolved to their most specific type before being returned to Python. This is not the case in C++ where a type may be returned and then require additional downcasting to get a more specific type. This may provide a more *pythonic* interface, but the user should be aware that the return types may not exactly much the C++ documentation, although since they will be a sub-class they should have the same functionality. For example, copying a line in PythonOCC may have looked like:

```
handle_geom = line.Copy()
new_line = Handle_Geom_Line.Downcast(handle_geom).GetObject()
```

where in pyOCCT it will now look like:

```
new_line = line.Copy()
```

with `new_line` being of type `Geom_Line`. There are no more `Handle_*` types available to import or use.