Python API: Basics

1. TiGL Workshop, September 11 / 12, Cologne

Martin Siggel
German Aerospace Center

DLR

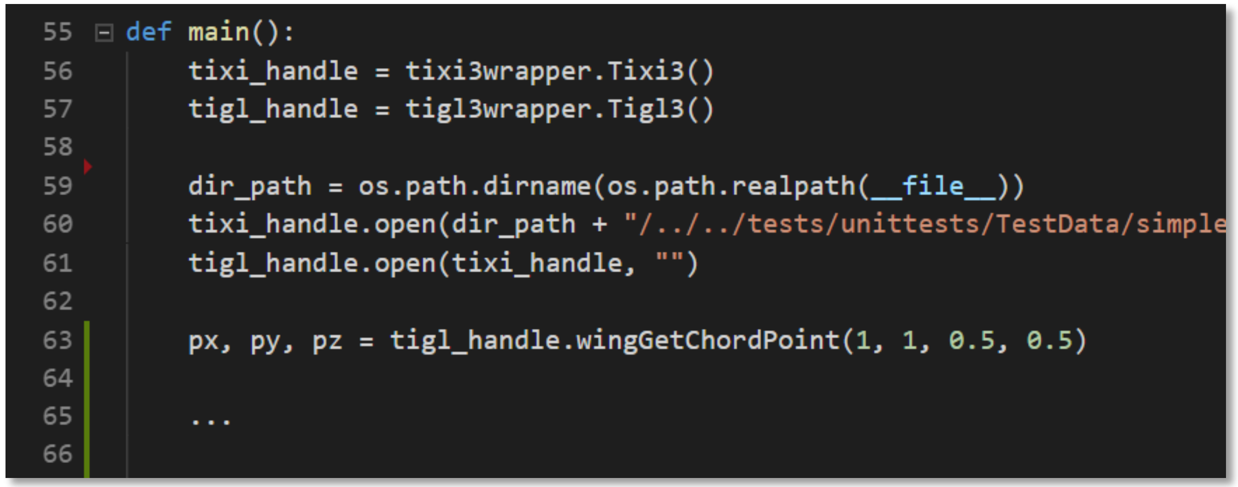
Outline

- Motivation: Why a new low-level Python API?
- Installation
- Design and architecture
- CPACS tree traversal with the tigl3.configuration module
- Geometric components and named shapes
- File exports with tigl3.exports
- Practical session

Motivation
The old “high-level” API

- Wraps the TiGL C API to Python
- Used for many years by most of our users
- Pros:
  - Very easy to use
  - Hides internal complexity of TiGL
- Cons:
  - Can do only what developers intended
  - Very limited functionality
  - No direct interaction with geometries
  - Can be slow

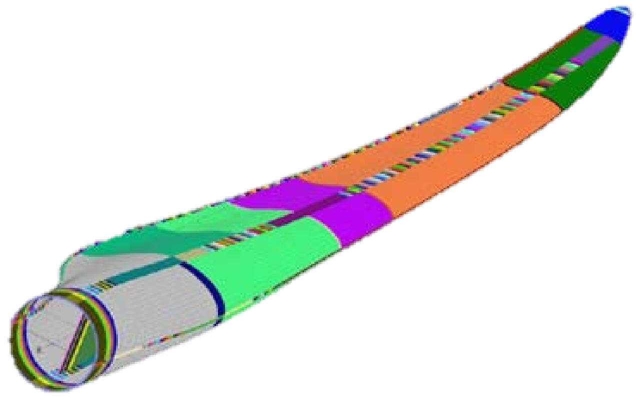
➢ We need a way, to access TiGL’s internals



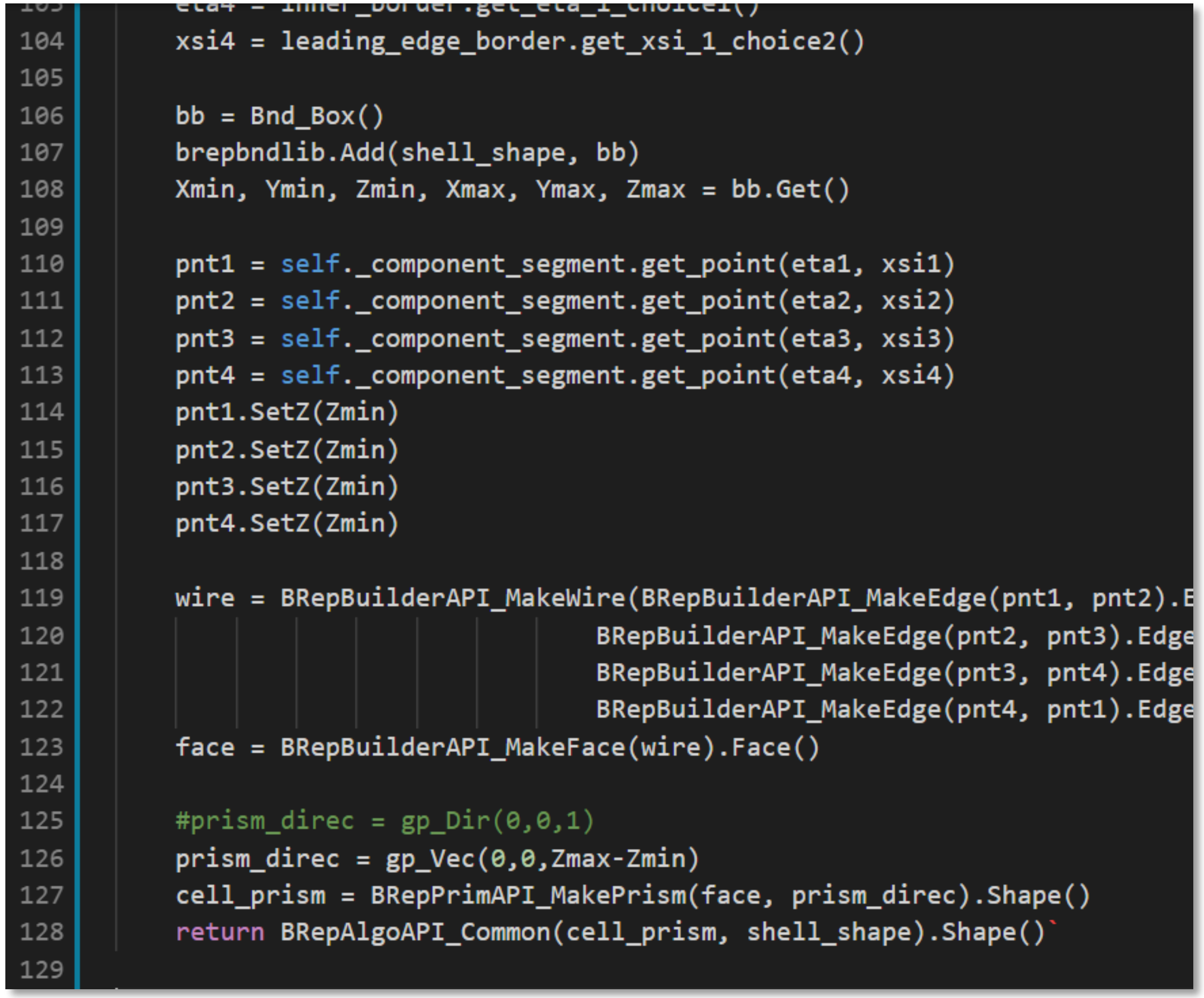
```python
def main():
    tixi_handle = tixi3wrapper.Tixi3()
    tigl_handle = tigl3wrapper.Tigl3()

    dir_path = os.path.dirname(os.path.realpath(__file__))
    tixi_handle.open(dir_path + "/../../tests/unittests/TestData/simple
    tigl_handle.open(tixi_handle, "")

    px, py, pz = tigl_handle.wingGetChordPoint(1, 1, 0.5, 0.5)

    ...
```

Motivation
An example

- Problem:
  - Colleagues wanted to create wing cell geometries with different materials for FEM simulations of wind wheels
  - Wing cells not modelled in TiGL



- Solution:
  - They used TiGL low-level python API with pythonOCC
  - They created their own cell geometry code



```
        eta4 = inner_border.get_eta_1_choice1()
        xsi4 = leading_edge_border.get_xsi_1_choice2()

        bb = Bnd_Box()
        brepbndlib.Add(shell_shape, bb)
        Xmin, Ymin, Zmin, Xmax, Ymax, Zmax = bb.Get()

        pnt1 = self._component_segment.get_point(eta1, xsi1)
        pnt2 = self._component_segment.get_point(eta2, xsi2)
        pnt3 = self._component_segment.get_point(eta3, xsi3)
        pnt4 = self._component_segment.get_point(eta4, xsi4)
        pnt1.SetZ(Zmin)
        pnt2.SetZ(Zmin)
        pnt3.SetZ(Zmin)
        pnt4.SetZ(Zmin)

        wire = BRepBuilderAPI_MakeWire(BRepBuilderAPI_MakeEdge(pnt1, pnt2).E
                                       BRepBuilderAPI_MakeEdge(pnt2, pnt3).Edge
                                       BRepBuilderAPI_MakeEdge(pnt3, pnt4).Edge
                                       BRepBuilderAPI_MakeEdge(pnt4, pnt1).Edge
        face = BRepBuilderAPI_MakeFace(wire).Face()

        #prism_direc = gp_Dir(0,0,1)
        prism_direc = gp_Vec(0,0,Zmax-Zmin)
        cell_prism = BRepPrimAPI_MakePrism(face, prism_direc).Shape()
        return BRepAlgoAPI_Common(cell_prism, shell_shape).Shape()
```

The Low-Level Python API

Why?

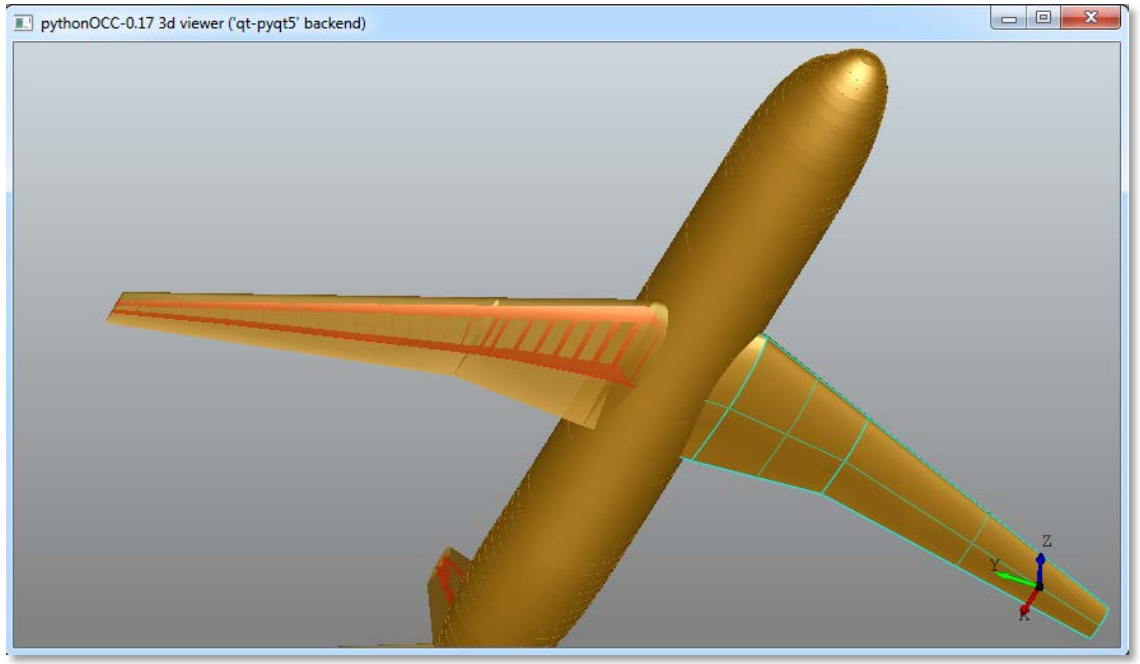
- We want to give you a flexible tool to be creative
- You are the experts: You should build your own models!

What do you get?

- Fast object-oriented access to all TiGL's internals without compilation
- Full interoperability with OpenCASCADE (pythonOCC)
- Flexibility
- New geometry operations like Gordon surfaces
- Customizable file exports and imports
- Visualization with only a few code lines
- Create meshes using GMesh or SMesh

What do we get?

- We can focus on internals and algorithms
- Maybe more feedback or even contributions



Created with low-level API. ~50 lines of code

Installation
Conda

- Conda: „*Package, dependency and environment management for any language—Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN”*

- De-facto standard in scientific community

- Differences to PIP:
    - Packaging also non-Python packages like libraries, executables …
    - Support for compiled binary packages
    - Custom package repositories

- Common to PIP:
    - Virtual environments
    - Large package repositories
    - Update of packages

TiGL’s Conda repository: https://anaconda.org/DLR-SC/

Installation
Conda

1. Install TiGL 3 into **new virtual environment** „tigl_env“ using python 3.5:

```
>> conda create -n tigl_env tigl3 python=3.5 –c dlr-sc
```

2. Switch into tigl_env environment:

```
>> activate tigl_env
```

(windows)

```
>> source activate tigl_env
```

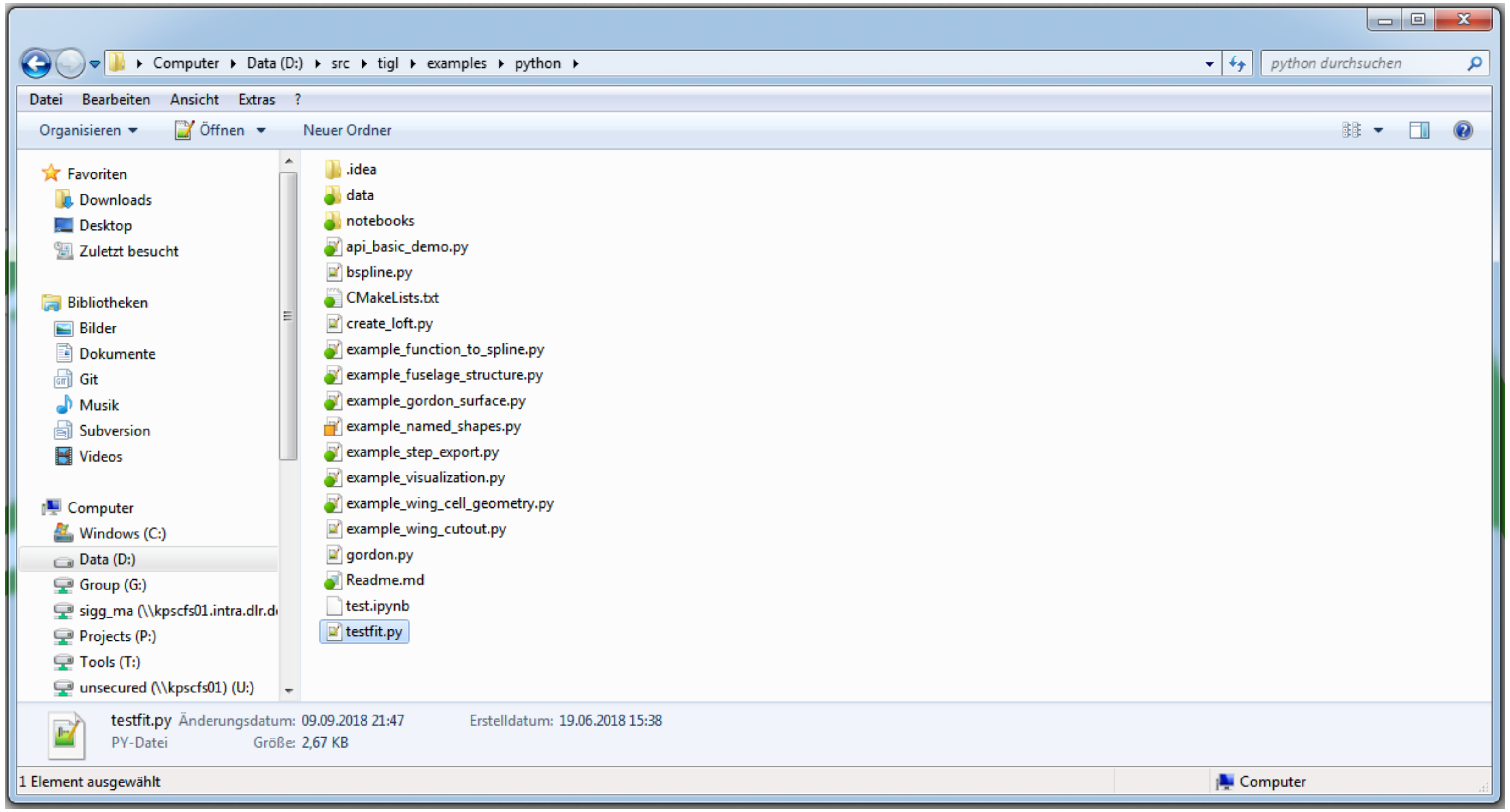
(mac/linux)

3. There, install other packages or run python interpreter, e.g.:

```
>> conda install numpy matplotlib
```

Getting started

- Have a look at the examples/python directory of TiGL!



.idea
data
notebooks
api_basic_demo.py
bspline.py
CMakeLists.txt
create_loft.py
example_function_to_spline.py
example_fuselage_structure.py
example_gordon_surface.py
example_named_shapes.py
example_step_export.py
example_visualization.py
example_wing_cell_geometry.py
example_wing_cutout.py
gordon.py
Readme.md
test.ipynb
testfit.py

testfit.py Änderungsdatum: 09.09.2018 21:47 Erstelldatum: 19.06.2018 15:38
PY-Datei Größe: 2,67 KB

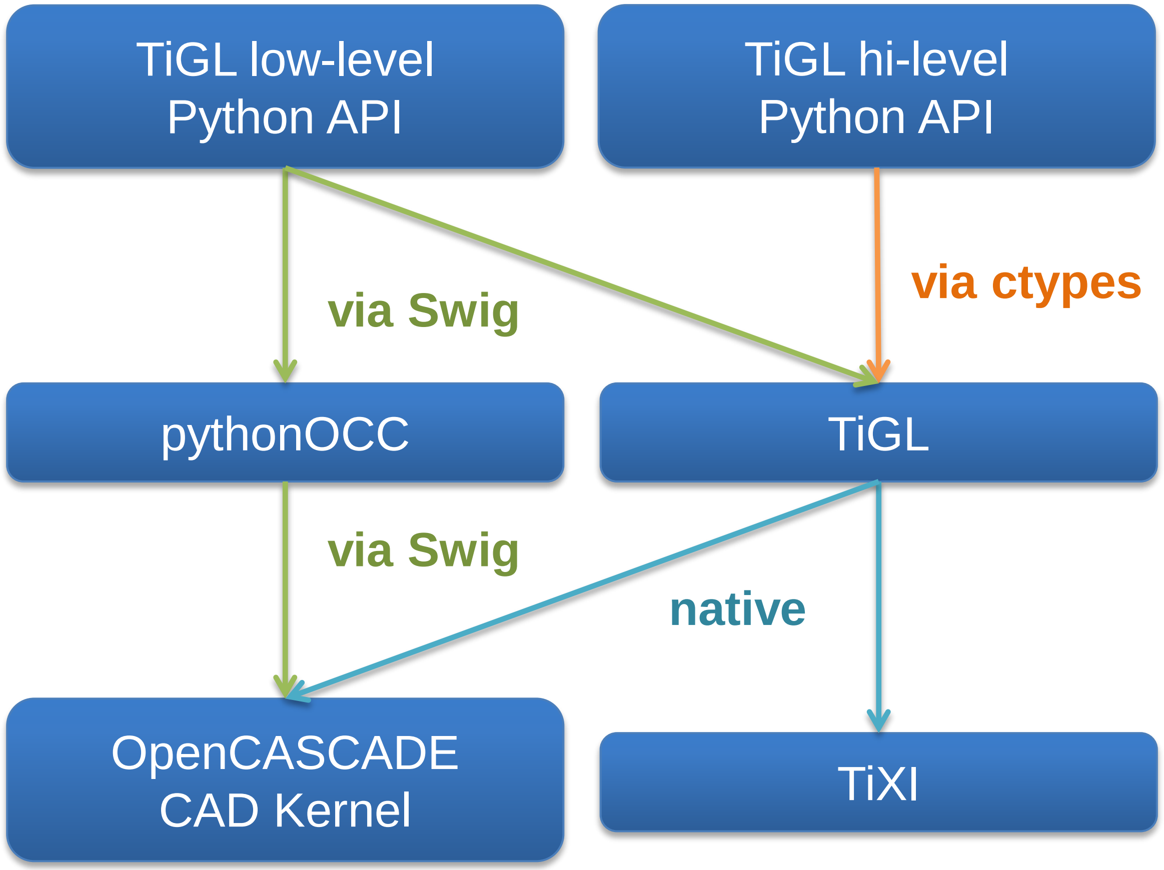
1 Element ausgewählt Computer

Design philosophy

- Direct unaltered **access to the internal C++ API**
    - Gives the same possibilities as if you were using the C++ interface
- Full **interoperability with OpenCASCADE**
- Provide **additional wrapper** functions to make use **more pythonic**
- Small drawback: **C++ API is not stable** and sometimes changes (a bit)

Architecture

- OpenCASCADE bindings **pythonOCC** generated with SWIG software: http://www.swig.org/
- TiGL low-level API also generated with SWIG from TiGL and pythonOCC
- Objects between pythonOCC and TiGL can be exchanged
- Input files for Swig can be found in: `<TiGL-Source>/bindings/python_internal`
- SWIG can also generate bindings for other languages!
- Small drawback: Bindings must be compiled for specific platform and python version!



Internal Structure

- Low-Level API consists of the following modules:

Module	Description
`tigl3.configuration`	Access to the whole cpacs tree including wings, fuselages …
`tigl3.geometry`	Functions related to geometry operations, e.g. curve interpolation
`tigl3.curve_factories`	Functions to create curves based on `tigl3.geometry`
`tigl3.surface_factories`	Functions to create surfaces based on `tigl3.geometry`
`tigl3.occ_helpers`	Pythonic wrappers to OpenCASCADE functions
`tigl3.exports`	File export classes
`tigl3.imports`	File import classes
`tigl3.boolean_ops`	Classes to perform Boolean operations
`tigl3.tmath`	A few math functions
`tigl3.core`	Core functionality, like error handling

The tigl3.configuration module
CPACS tree traversal

- TiGL handles several aircraft configurations at a time. To get a specific one, use the **configuration manager class:**

```
from tigl3 import tigl3wrapper                                          0
from tigl3.configuration import CCPACSConfigurationManager_get_instance

tigl_handle = tigl3wrapper.Tigl3()
tigl_handle.open(tixi_handle, "")
mgr = CCPACSConfigurationManager_get_instance()                          1
aircraft_config = mgr.get_configuration(tigl_handle._handle.value)       2
```

0 Load the „old“ high-level python API

1 Retrieve the global **CCPACSConfigurationManager** object

2 Access a **specific aircraft** config, given by it‘s TiGL handle (integer value)

- `aircraft_config` is instance of class CCPACSConfiguration. Gives direct access to fuselages, wings, systems …

The tigl3.configuration module
CPACS tree traversal

- Use `help` to look for the available methods and members of each CPACS node

```
help(aircraft_config)
```

- Examples:

```
wing = aircraft_config.get_wing(1)
wing = aircraft_config.get_wing("wing_uid")
wing.get_uid()
segment = wing.get_segment(2)
segment = wing.get_segment("segment_uid")
aircraft_config.get_fuselage(1)
aircraft_config.get_uid()
```

- Access wing spar properties:

```
aircraft_config.get_wing(1).get_component_segment(1).get_structure().get_spar_segment(1)
     .get_spar_cross_section().get_rotation()
```

- This way, accessing a specific object can be lengthy!

The tigl3.configuration module
UID Manager

- Most objects that have a CPACS uid are registered at the **UID manager** when reading the CPACS file
- The UID Manager is **a member of the CCPACSConfiguration** object
- The UID Manager enables **direct access** to those objects:

```
uid_mgr = aircraft_config.get_uidmanager()
wing  = uid_mgr.get_geometric_component("WingUID")
spars = uid_mgr.get_geometric_component("SparSegmentsUID")
…
```

- Caveat: From Python **only geometric components** can be queried at the moment

The tigl3.configuration module
Geometric components

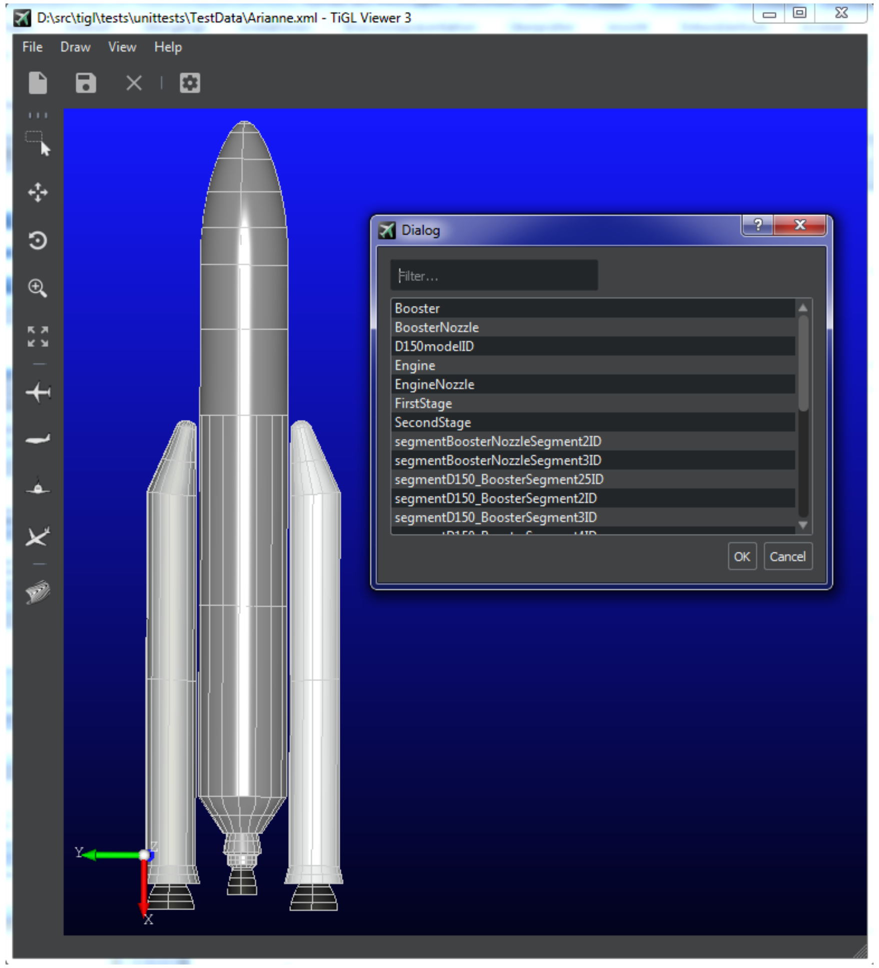
- Cpacs nodes that have a geometry are a geometric component and **derived from class ITiglGeometricComponent**

- To look, which geometric components currently are supported:
  1. Open TiGL Viewer
  2. Menu → Draw → Aircraft → Draw any component

- These are today:
  - Wings, Wing Segments, Fuselages, Fuselage Segments, Component Segments, Wing Cells, Wing Spars, Wing Ribs, Wing Chord Face, Wing Upper/Lower Side, External Components, Systems, Rotors, Rotor Blades, Fuselage Cross Beams, Fuselage Cross Beam Struts, Long Floors, Pressure Bulkheads, Doors

- Access the geometry of geometric componenent with **get_loft()** method:

```
wing.get_loft()
```


D:\src\tigl\tests\unittests\TestData\Arianne.xml - TiGL Viewer 3

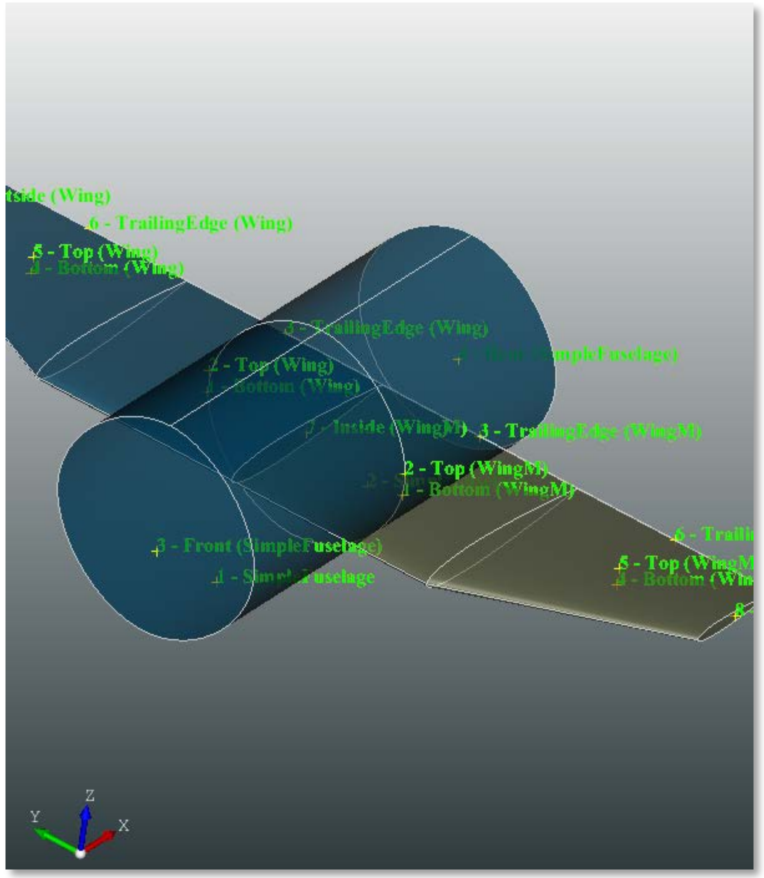
Dialog

Filter...

Booster
BoosterNozzle
D150modelID
Engine
EngineNozzle
FirstStage
SecondStage
segmentBoosterNozzleSegment2ID
segmentBoosterNozzleSegment3ID
segmentD150_BoosterSegment25ID
segmentD150_BoosterSegment2ID
segmentD150_BoosterSegment3ID

Named Shapes
Shapes with metadata

- `component.get_loft()` returns an instance of tigl3.geometry.CNamedShape
- Contains additional metadata next to the raw shape
- Is used by the TiGL exports to write also identifiers and names
- Allows to track the history of shape creation and modification, in particular after Boolean operations
- Contains a graph of the shape creation history



Named Shapes
Shapes with metadata

Methods of tigl3.geometry.CNamedShape object are:

- `.shape():` The actual shape as used by OpenCASCADE (OCC.TopoDS.TopoDS_Shape)
- `.name():` The name of the shape (string)
- `.short_name():` A short name of the shape (used for IGES identifiers)
- `.get_face_traits(i):` properties of the i-th face, including:
  - `.name():` Name of the face
  - `.set_name():` Set the name of the face
  - `.origin():` Shape from which the face was originally created from (CNamedShape)

Named Shapes
Creation and modification

- Build from an OpenCASCADE shape, e.g.

```
from tigl3.geometry import CNamedShape
box = BRepPrimAPI_MakeBox(0.5, 0.5, 0.5).Shape() # Build a box with OpenCASCADE
box_ns = CNamedShape(box, "MyBoxObj")
```

- Get number of faces of the box

```
print(box_ns.get_face_count())    # Should be 6
```

- Print current name of the 4-th face

```
print(box_ns.get_face_traits(4).name())
```

- Modify the name of the 4-th face

```
box_ns.get_face_traits(4).set_name("bottom")
```

The tigl3.export module

- File exports are interfaces to simulation software or CAD programs
- The tigl3.export module includes exports for the following file formats

Format	Description
Step (ISO 10303)	Current industry standard for CAD products.
IGES	Predecessor to Step. Standard file exchange format for CAD surfaces.
BRep	Native OpenCASCADE format to store shapes.
VTP (VTK Polydata Mesh)	Triangulated surface mesh for Paraview / VTK. Can include metadata.
STL	Simple surface mesh format. Used for 3D printers.
Collada	Standard format for 3D rendering software like Blender.

The tigl3.export module
CTiglCadExporter Interface

- All exporters have a common interface CTiglCADExporter

- The most important methods are

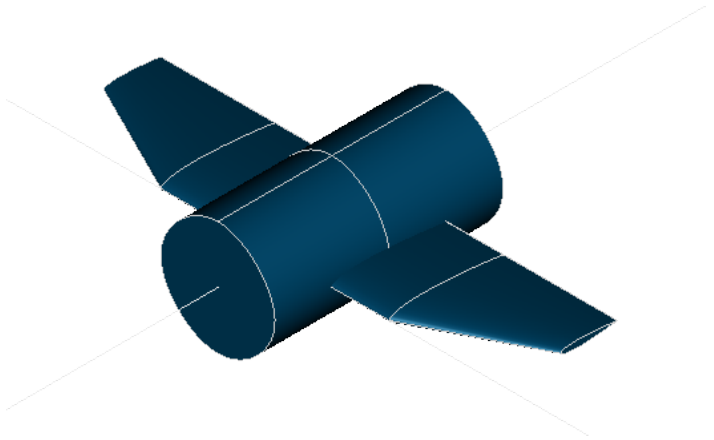
Method	Description
add_shape(named_shape, options)	Adds a (named) shape to the file. Options are exporter specific, e.g. IGES level, triangulation accuracy …
add_configuration(aircraft_config, options)	Adds the whole aircraft to the exporter
add_fused_configuration(aircraft_config, options)	Adds the fused aircraft to the exporter by trimming all surfaces first.
write(filename)	Writes the geometries to the specified file.
supported_file_type()	Returns semicolon separated list of supported file extensions, like “igs;iges”

The tigl3.export module

Example: Basic IGES Export

- Write left and right wing + fuselage to test.iges:

```
import tigl3.exports
exporter = tigl3.exports.create_exporter("igs")
exporter.add_shape(wing.get_loft())
exporter.add_shape(wing.get_mirrored_loft())
exporter.add_shape(fuselage.get_loft())
exporter.write("test.igs")
```

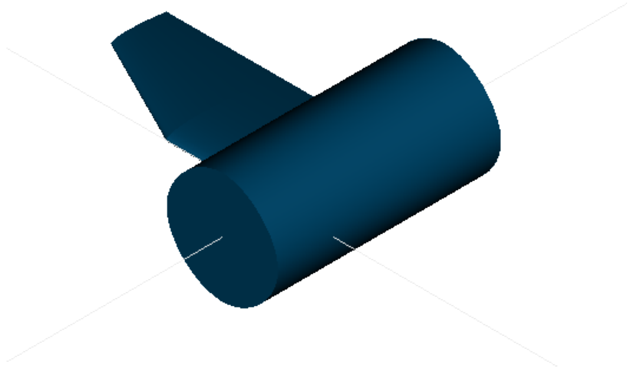


The tigl3.export module
Example: IGES Export with layers

- The IGES file format supports layers that are used e.g. by mesh generators
- Write fuselage and wing to separate layers 1111 and 2222:

```python
import tigl3.exports
exporter = tigl3.exports.create_exporter("igs")
exporter.add_shape(fuselage.get_loft(), tigl3.exports.IgesShapeOptions(1111))
exporter.add_shape(wing.get_loft(), tigl3.exports.IgesShapeOptions(2222))
exporter.write("test_layers.igs")
```

```
                                                                        S0000001
,,31HOpen CASCADE IGES processor 6.8,13HFilename.iges,                  G0000001
16HOpen CASCADE 6.8,31HOpen CASCADE IGES processor 6.8,32,308,15,308,15,G0000002
,1.,2,2HMM,1,0.01,15H20180910.100150,0.000195405,2000.322906,4HTiGL,    G0000003
33HGerman Aerospace Center (DLR), SC,11,0,15H20180910.100150,;          G0000004
     514       1       0       0    1111       0       0       000010000D0000001
     514       0       0       1       1                      F1        D0000002
     510       2       0       0    1111       0       0       000010000D0000003
     510       0       0       1       1                F1              D0000004
     .
     .
     514      90       0       0    2222       0       0       000010000D0000079
     514       0       0       1       1                      W1        D0000080
     510      91       0       0    2222       0       0       000010000D0000081
     510       0       0       1       1                Bottom          D0000082
     128      92       0       0       0       0       0       000010000D0000083
```



The tigl3.export module

Example: IGES Export of whole aircraft with custom settings

- Now lets export the whole aircraft.

- Lets tweak the default settings:
  - By default, the IGES export includes the far field and produces does not include symmetries

  - Lets change this

```
import tigl3.exports
# get the current IGES settings and modify them
iges_config = tigl3.exports.get_export_config("igs")
iges_config.set_include_farfield(False)
iges_config.set_apply_symmetries(True)

# create the exporter with the new settings
exporter = tigl3.exports.create_exporter("igs", iges_config)
exporter.add_configuration(config)
exporter.write("test.igs")
```

The tigl3.import_export_helper module
Convenience layer

- Wrap the exporters in simple to use functions
- Currently only one function: `export_shapes(shapes, filename, deflection=0.001)`
  - Automatically determines exporter according to file extension
  - Can also use OCC.TopoDS.TopoDS_Shapes
  - Example:

```
from tigl3.import_export_helper import export_shapes

export_shapes([a_named_shape, a_topods_shape], "mystepfile.stp")
```

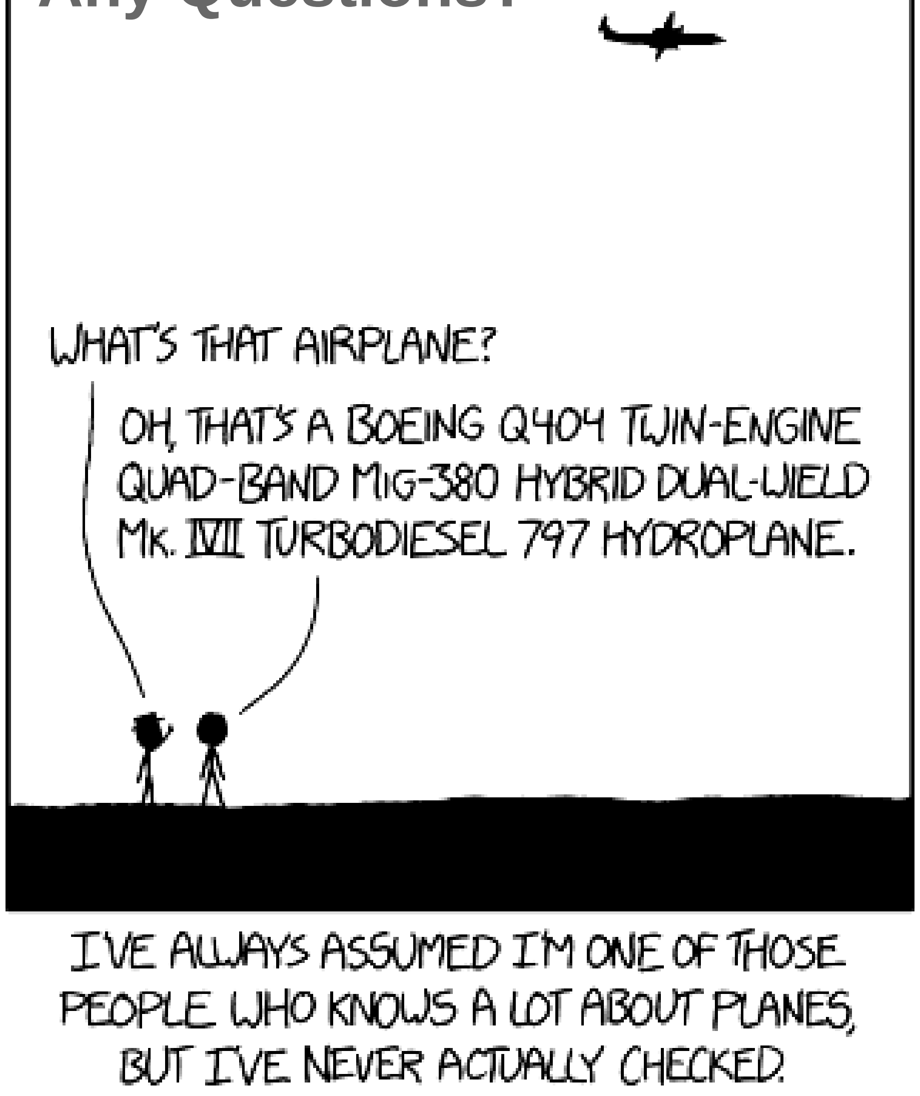
Practical Session

- Now it‘s your turn

- Goal: Get to know the new API. Look around. Play around.

- Possible Tasks:
  1. Install tigl3 via conda if not yet done
  2. Open exercise 1 from course material inside a jupyter notebook
  3. Load an aircraft configuration with TiXI and TiGL
  4. Access the first wing
  5. Check, which methods the CCPACSWing class offers
  6. Print the wing UID
  7. Get the shape of the wing
  8. Display all face names of the wing
  9. Rename the trailing edge of the wing
  10. Create named shape with a box
  11. Export wing and box to an IGES file with separate layers
  12. Drop exported *.igs file into TiGL Viewer to check.
  13. Open IGES file in text editor. What is in there?

Any Questions?


WHAT'S THAT AIRPLANE?
OH, THAT'S A BOEING Q404 TWIN-ENGINE QUAD-BAND MIG-380 HYBRID DUAL-WIELD MK. XVII TURBODIESEL 797 HYDROPLANE.
I'VE ALWAYS ASSUMED I'M ONE OF THOSE PEOPLE WHO KNOWS A LOT ABOUT PLANES, BUT I'VE NEVER ACTUALLY CHECKED.