UNIVERSITY OF PADOVA

SCHOOL OF ENGINEERING

MASTER'S DEGREE IN ICT FOR INTERNET AND MULTIMEDIA

PYTHON API FOR ALTAIR INSPIRE STUDIO WITH FUNCTIONALITY OF CAPTURING 3D MODELS FROM RGBD SENSORS

Advisor: Prof. Pietro Zanuttigh
Coadvisor: Giampaolo Pagnutti

Student: Pier Angelo Vendrame
Student id 1171679

Padova, October 14th, 2019

Academic Year 2018-2019

Abstract

The use of 3D modeling software is more and more important for the industry. It offers new approaches to the design tasks and new perspectives to look at projects, in a more realistic and easier way than 2D.

During the years, several software suites have been developed; some have specialized for particular purposes, whereas some others have remained generic, and offer a way to provide specific features, or to integrate with the rest of productive process, through plugins.

This thesis shows how we implemented the Python support to write plugins and to interact with Altair Inspire Studio, in addition to the C++ SDK they have been offering for years.

Then, the thesis shows how we used this novel API to develop a plugin to capture 3D models from reality, using RGBD sensors.

Contents

Part I: Introduction

Chapter 1

Contextualization

Several industries have switched from 2D to 3D modeling, and more are transitioning, because the latter has several advantages.

First, it is more efficient, as the model is unique, and the changes need to be performed once. Moreover, many programs allow to project the 3D model to obtain 2D representations.

Secondly, 3D helps to visualize objects, which is more difficult to do from 2D draws, especially for non-technical personnel.

In addition to that, software that implements construction history allows to edit a portion of the model and have the changes applied in cascade to all the linked parts.

Many 3D modeling programs provide APIs to write plugins.

A reason is to extend the file formats they can handle. In this way, 3D modeling software can be inserted in a pipeline: with plugins, users can open files created in previous stages, modify them, and then export them in a format to proceed with further steps.

In many cases, plugins allow also to extend the 3D software itself, by adding new functionalities, operations, custom commands and so on.

Traditionally, C and C++ were the programming languages used for these purposes, because they have high performances and they allow to directly interact with the hardware. However, the use of script languages for these purposes, in particular Python, has been increasing.

The support for Python plugins and scripting in Altair Inspire Studio is one of the main topic of this thesis. The other one is how we used it to write a plugin that allows to scan 3D objects from reality using RGBD sensor, an inexpensive way of doing reverse engineering, which is the creation of a 3D model starting from an existing physical realization.

This practice began during 1990s, using LiDAR, that are laser-based scanners, very precise, but still expensive nowadays.

Some of the techniques developed for these devices are used also with small,

consumer grade RGBD sensors (color cameras that include also a depth channel), which started spreading around 2010: their precision is inferior, but in some cases it is sufficient and the low cost factor prevails.

Chapter 2

Altair Inspire Studio

3D modeling software has several possible classifications.

One is *history-based*, or *parametric* modeling, and *direct* modeling. The former allows the user to act on parameters of objects, such as lengths and angles. It is also called history-based because when the user modify these data, the changes are applied in cascade to all the linked or dependent parts of the scene. This feature helps to create families of products, and, in general, parametric modeling is structured and allows to have precise definitions.

On the other hand, direct modeling allows to manipulate the geometries of the models, rather than edit their features. It is considered somehow more flexible, but it makes impossible to go back to previous stages, modify the model and have all the rest of the changes applied again in chain; in other words, every change is definitive.

Another classification is about the geometry representation used. *Boundary Representation* or *B-Rep* is a method for representing shapes using their limits, and can be used to represent exact geometries, like spheres. A different representation is based on polygons: it uses vertex positions and connections between them. Some geometries may be only approximated in this way, and in general better approximations require a higher number of vertices. In particular, it is very common to use triangular representation, although quad-based representations are also remarkable; NURBS (*Non-uniform rational B-spline*) are a common mathematical model for B-Rep, instead.

Altair Inspire Studio is a 3D modeling and rendering software developed by Altair Engineering for designers. It allows free-form surface design, solid modeling based on NURBS, and polygonal modeling with n-side polygons. All the objects in the 3D scene are controlled by parameters and by construction history.

It is the new version of the program formerly known as solidThinking Evolve, which has been used by many companies and individual artists to design several kind of products.

Although Inspire Studio is a software rich of features, Altair also offers the possibility to extend it, with a SDK, which they call PDK, or *Plugin Development Kit*. Using this kit, developers can add functionatilities to the software: they can add *modeling tools*, importers and exporters, with the same capabilities that integrated tools would have.

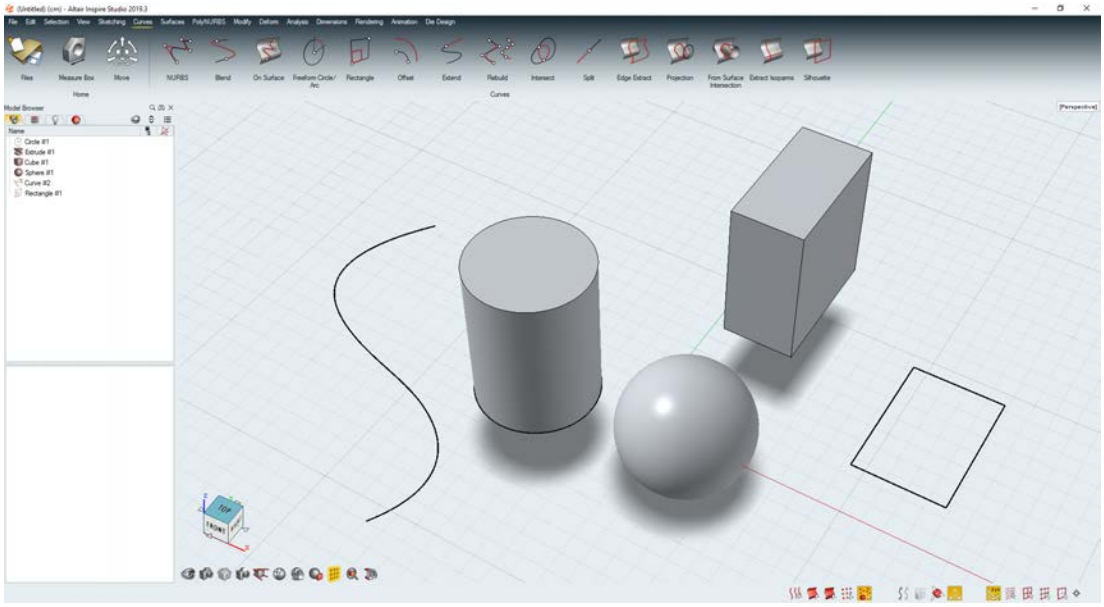


Figure 2.1: A screenshot of Altair Inspire Studio.

Modeling tool is an Inspire Studio specific term that denotes any tool that can be used to create objects and modify existing ones. Every button available on the ribbon is linked to a modeling tool, an even the "move" and the "visualization color" are special kinds of modeling tools.

Figure 2.2: All these icons on the ribbon bar are modeling tools, including the *move* and the *measure box*.

There are many reasons to write custom commands. One might develop a tool that outputs shapes often used in some applications, either to reduce the time that would take to create them from simple geometries, or to link them to some specific parameters.

As an example, one might develop a tool to create screws and threaded rods, taking as input parameters the thread, the diameter and the length, or have some presets for some commercially available items. A tool like this might even interface to a database of products that are already in the inventory.

Indeed, integrating Inspire Studio with other services and tools used in a company is another reason to write plugins: interfacing with computer-aided

manufacturing software, or creating and exporting a toolpath for a CNC router are only some examples of what programmers can do.

Finally, the PDK is made especially for third parties, but having a robust plugin API allows also specialized first party development teams to build tools for Inspire Studio, without knowing how the kernel of the software works. For example, a team for tools to work on metals, another team specialized in 3D printing and so on.

Part II: The Python PDK

Chapter 3

Motivations

Python is a scripting language developed since 1991, first by Guido van Rossum and now by the independent and no-profit Python Software Foundation.

It is considered very easy to learn; it focuses on code readability and on constructs and structures that help programmers to write clear code and organize projects efficiently and in an ordered fashion.

These are some of the reasons for which Python spread also to people that traditionally were not involved in software development. Nevertheless, this language is appreciated also by experienced programmers and used in some of the biggest IT companies [29].

Being an interpreted programming language with dynamic typing, Python is much slower than compiled languages [10]. However it features a C API [28], to write some "glue code", called *bindings*, to any library exposing a C interface, to allow scripts to use it, as we will see on chapter 5.

This is exploited by some libraries, such as NumPy to provide a Python API while performing all the heavy computations using native highly efficient code, that can exploit advanced processor features such as SIMD extensions [31]. Another example is the TensorFlow machine learning framework, that features a Python API to build models, that then are trained on GPUs using Nvidia CUDA.

The C API of Python also allows to embed it, even in commercial software, thanks to its permissive open source license [26], and this is what has been done in Inspire Studio.

There are many reasons to do that. The first one is the will to **enlarge** the possible plugin developer audience and **simplify** the development.

The PDK is written is distributed as a collection of C++ header and library files, with the documentation to use them. C++ is a lower level language; people who wish to code in this language need to have a solid programming background. Moreover, they need to have a C++ compiler or C++ development environment, and the technical skills required to use them. On the contrary, writing a Python plugin is easier and requires only a text editor, although more advanced tools are helpful.

The lack of compilation process and of a formal project structure represents another reason to have a Python PDK: it enables *rapid prototyping* and other rapid development approaches. One might, for example, write a Python plugin to isolate, develop and test a part of a bigger plugin, which may be written either in Python or in C++.

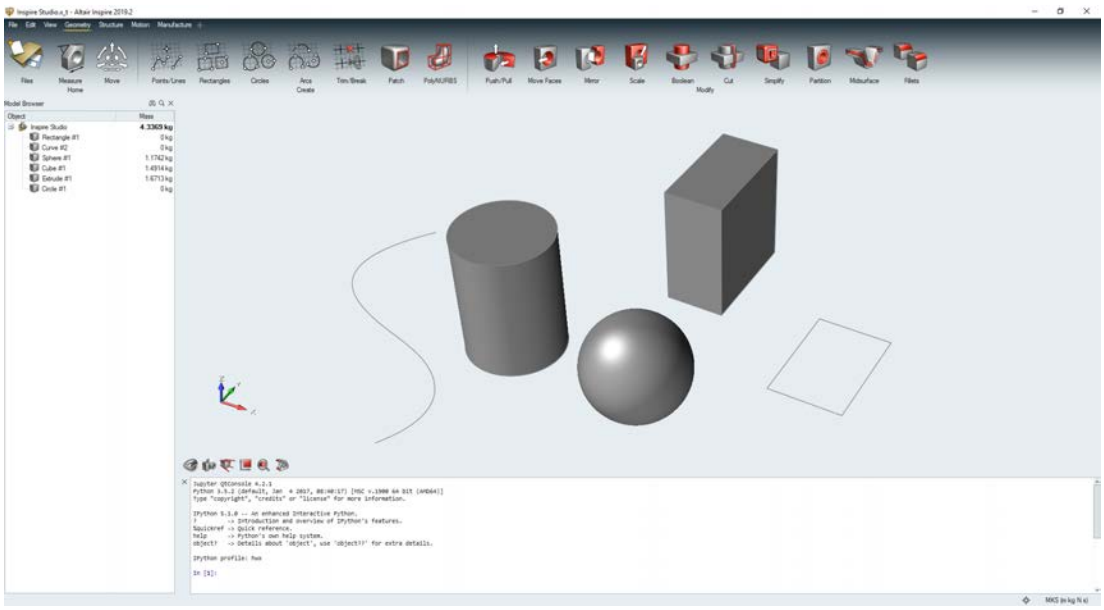


Figure 3.1: The Python console in Altair Inspire, another software from the Inspire lineup. The plan is to add a similar console also to Inspire Studio.

Another reason to develop the Python bindings is that they could provide a **programming interface** to Inspire Studio, either with a console, or by running scripts. Adding this feature would not require neither additional work for binding developers, nor additional knowledge for users, because the bindings would be unique, they do not need to be differentiated for the function they are used for. The only difference is how the main program runs users' code, i.e. by loading automatically some files at the startup, or by having a file dialog asking the user which script they want to execute, or by waiting for keyboard events on the console.

Python has also some disadvantages. It is very slow [10], and in some cases having a compiled support library is not a chance; in this case a C++ plugin is the only choice. Another problem is that, although Python supports multi-threading in its standard library, the reference implementation of the interpreter is inherently single threaded, however there is no official support for concurrency in Inspire Studio plugins, yet.

In any case, the Python PDK is supposed to be an **alternative** to the C++ PDK, not a replacement.

Chapter 4

Requirements and constraints

Like all projects, also ours has some requirements and constraints: we are explaining them in this chapter, along with some additional information on the project itself.

4.1 Objectives

Inspire Studio PDK was the starting point for the bindings. It is written in C++, and it uses several constructs and patterns of the object-oriented programming (OOP).

We may divide it in three parts:

1. a library of **basic utilities**, such as vectors and other math helpers, I/O and so on;
2. the **interfaces** to interact **with the program**, its UI, its functionalities, etc;
3. a library to **create and modify geometries** represented as *B-Rep* (boundary representation).

Together, they are a few hundreds of classes, with the second part being many times bigger than the other two.

The initial requirement for the Python port was to cover the interfaces to create modeling tools and manage their parameters, the library for the B-Rep geometries, and the utilities that these classes require, such as 2, 3, 4 dimensional vectors and 4×4 matrices used for transformations in homogeneous coordinates.

4.2 Constraints

The interfaces of Inspire Studio (the ones at point 2 of the previous list) comprise much more than that, for example there are also classes to customize the photorealistic rendering, classes for the 2D sketching, classes to manage polygonal geometries and much more. These sections were not required to work at the initial phase of the project: we prioritized testing how the aforementioned parts could be bound accordingly to some requirements and constrains.

4.2.1 Separation from the core

It has been decided that the bindings should not be part of the Inspire Studio core.

One reason is that any problem with the bindings should not block the development of the rest of the software.

We might consider as part of this constraint also the fact that the bindings should use only code available in the PDK, without using additional core code.

Finally, we aimed at having the bindings as *transparent* as possible to the rest of developers, i.e. modifying large or frequently used portions of the code base was not an option.

We encountered some difficulties due to the inherent differences in the languages, e.g. methods accepting a C array, i.e. pair of arguments with pointer to data and number of elements. Python, as many high level languages, does not have the concept of pointer, but we did not modify these methods to receive a C++ container, even though it would have been easier to interface to, with bindings, because some workarounds existed and changing those functions would have been a huge problem for the rest of the program.

However this was not a strict constraint, and eventually some small modifications have been made, especially if it could fix some bugs or improve the quality of the C++ code itself.

For example, some methods had accepted parameters as C varargs, which is a problem, because the C standard leaves some freedom of implementation to compilers, making it more difficult to convert call arguments from Python to C for bindings. Therefore, those methods were converted to accept a `std::initializer_list` instead.

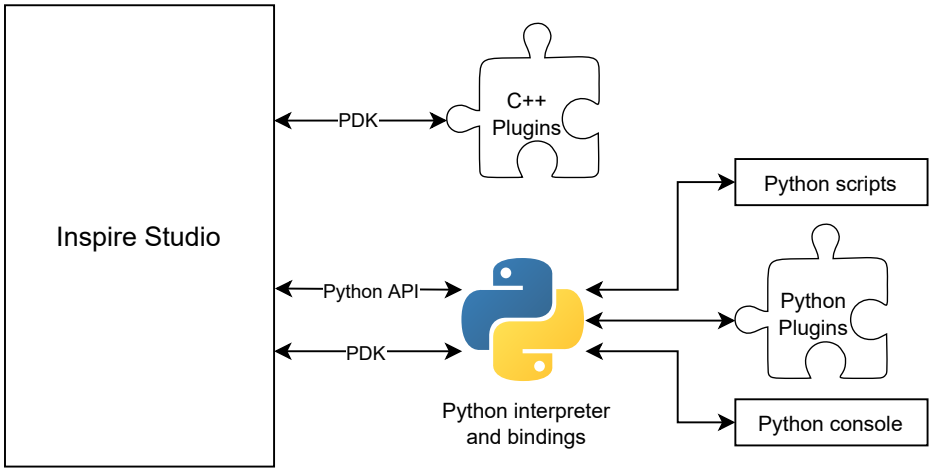


Figure 4.1: The plugin architecture and the role of Python bindings in it.

4.2.2 Automation of the process

Binding the code manually, even in the restricted case, would take a very long time, because there are too many classes to pass. Therefore we wanted to have an automated mechanism.

As a worst case, we wanted to have at least something to run once to have the most of the code bound, and then slowly refined manually.

Moreover, a part from a handful classes, the PDK is not standalone code created only for plugins (e.g. some wrappers to the core code), but it is *extracted* from the main codebase. The internal developers can mark code to make it available for the PDK; because of that, some parts are very likely to change often and quickly.

Therefore, the objective would be to have the automated tool able to run again on the changed code, without destroying any manual fix.

However, we wanted to avoid using tag/macro based systems added to the code to bind. First, because it would be against the transparency constraint, as the other developers should be instructed on how to use them; second, passing the code to add those tags would also take a long time.

This may seem in contrast with what is done for the C++ PDK, but it is not: the tags for the scripting language would be more difficult to add, as they should also take into account the language differences and try to smooth them out.

Chapter 5

Python C API

Python scripts need an interpreter to be run. The reference one is *CPython*; it is developed by the Python Software Foundation, using the C programming language, therefore the name.

CPython is released both as a standalone program and as a dynamic library. The latter exposes some of the interpreter functions and of its data structures as an API, to manage it, to extend it, and to execute Python code.

This chapter introduces some of the concepts of how the Python interpreter works and shows how to do some operations with this API.

5.1 The module system

Python is a modular language by design, and native extensions are managed in the same way as script units. *Modules* and *packages* are made available to a script or to an interpreter context, entirely or just parts of them, using the `import` statement.

The lookup of the item to load is made in this order [27]:

1. built-in modules and already loaded modules, which can be found in the `sys.modules` object within a Python script;
2. the directory containing the input script, or the current working directory when no file is specified, e.g. when running an interactive Python console;
3. a set of default paths that can be modified by changing the `sys.path` list within a Python script.

For the first case, the modules have to be registered by calling either the C function `PyImport_AppendInittab`, or the function `PyImport_ExtendInittab` before the interpreter initialization function is run. Therefore the creation of built-in module is useful especially for the embedding case, otherwise one would need to distribute a modified and recompiled version of the interpreter.

For cases 2 and 3, the *import machinery* looks for:

- `.py` files: the normal Python files;
- `.pyc` files: the compiled *bytecode* files that are generated from `.py` files as a cache to speedup the load of modules;

- directories with an `__init__.py` file: these are the *packages*, a way to group modules together;
- compiled modules: dynamic libraries, with the standard `.so` extension for POSIX systems, and the custom `.pyd` extension for Windows, although they are valid `.dll` files.

5.2 Essential concepts

The Python C API represents both the minimum requirement and the limits of native modules: everything a Python extension can do, can and needs to be done with this code.

All the other ways of creating extensions and bindings depend on it. Therefore, knowing its basics is very helpful, if not necessary, even when using a wrapper.

5.2.1 `PyObject`

`PyObject` is the most important data structure of this API. From the Python documentation [28]:

> it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a `PyObject`, but every pointer to a Python object can be cast to a `PyObject*`.

Although we are dealing with software programmed in C, we may think like it has been developed as OOP: in this scenario, we might think of `PyObject` as the parent class of everything.

It is the equivalent of `java.lang.Object` of the Java programming language, with the exception that, in Python, `PyObject` is the parent also of integers, of floating point numbers, of boolean values and even of the `None` reserved keyword.

Also pieces of Python code can be cast to a `PyObject *`: it is the case of modules, functions, type definitions, including classes (not only *class instances*, even the class *definitions* themselves are `PyObject`).

The type objects pointed by `PyObject`s, which are accessed through the macro `Py_TYPE`, since `PyObject`s are opaque pointers, give information about the properties of that object and the operations one can do with it, e.g. object members and methods.

The interpreter always checks these information when running a script, but also C extensions should do that, to avoid an undefined behavior, that might even lead to crashes.

5.2.2 Reference count

All Python objects are passed by *reference*, i.e. shared: assigning an object to a variable does not copy it, but makes the variable point to the same object and increases its reference count. When a variable goes out of scope, or when another value is assigned to it, the reference count is decreased.

Memory in Python is managed by the interpreter: the user does not need to destroy objects and free the memory they used, or *finalize* objects, in Python lexicon. The interpreter does the *garbage collection*: it periodically checks if some reference counts reach 0, and in case finalize these objects.

This is automatic in scripts, but from the point of view of a C extension, reference count increases and decreases need to be managed manually, and this will be the focus of the rest of this section.

Usually, when calling a function that creates and returns an object, the callee *transfers* the *ownership* to the caller, which means that the object reference count is already at least 1.

At this point, the caller does not need to do anything else, until it does not need the object anymore, in which case it has to decrease the reference count with the `Py_DECREF` macro. But if it passes the ownership, e.g. to return that object, it must not decrease the reference count.

When a function receives a parameter, it usually does not need to do anything: the callee *borrows* the reference ownership from the caller, and when it returns, it also returns the ownership. As an exception, when the callee needs to store that object somewhere, to access it after it returns, it must increase the reference count, using the `Py_INCREF` macro.

Sometimes, different ownership rules are used: it is important to always check the documentation of a function to understand how it works.

An object might want to store a copy of itself: to do this, it has to increase its reference count, the interpreter has a *cycle detector* to manage *circular references* without problems.

5.2.3 Exceptions and the `NULL` pointer

In Python, using exceptions in case of error is very common. Exceptions are even used for the interrupt signal (known also as `SIGINT`, or as Ctrl + C), and to

stop iterations on data structures.

C does not provide for the concept of exception, making another way of signaling error conditions needed.

Since many functions, both in the API and in the prototypes of the callbacks that developers have to specialize in their extensions, return a `PyObject` pointer, it has been decided that they should return `NULL` to report that they have encountered an error. Moreover, they should set an exception with functions like `PyErr_SetString`.

While going back on the call stack, functions must check if `NULL` was returned, and, in case, return it themselves, but they should not change the already set exception, unless they handled it and raised another one. Eventually, the interpreter will receive it and will look for the correct exception handler in the script. In case it does not find one, it may just output the error and the call stack on the *standard error* and kill itself.

Alternatively, a C function can handle exceptions using other functions starting with `PyErr_`, such as the one to get the type of exception and the one to clear the error status.

This means that `NULL` does not mean "no value" or "empty value", within the Python C API: functions should use the special `None` value, instead, managed with the singleton `Py_None` in C.

Some functions do not return pointers, but other integral types. In these cases, usually the -1 value is used instead of `NULL`.

5.3 Operations of a compiled module

Python modules do not have a strict, fixed structure. The initialization function is the only required function, programmers can choose how to implement the rest.

However, some operations are included in almost all modules.

5.3.1 The initialization function

This is the entry point of a compiled module. It is the only function that needs to be exported, and, when using C++, it needs to have C linking, i.e. it must be exported only with its function name, instead of the *mangled name*, the technique that C++ compilers use to encode additional information and solve problems such as overloads.

In particular, for modules in dynamic libraries, the name of this function should be in the format `PyInit_modulename`, and in Windows the function should

also be marked with the `__declspec(dllexport)` keyword. The API provides the `PyMODINIT_FUNC` macro to handle all these needs.

With built-in modules, these traits are not compulsory anymore, since the address of the initialization function is passed directly to the module registration functions as second parameter, however the macro is still useful, because it provides also information about the return type – a `PyObject *` in Python 3, `void` in Python 2.

The body of the initialization function comprises the check for validity of the structure contents, the creation of the module and the initialization of its attributes, including new types. Listing 10 shows this with actual C code.

5.3.2 Creating a new type

In an OOP context, the possibility of creating new types is fundamental. Python allows to do that by creating a custom structure to hold the data, and by populating a `PyTypeObject` struct with the type information, which include:

- `tp_members`, the table with the members of the data structure: it can be used with numeric types (`int`, `long`, `double`, etc), and with `PyObject *`, although it cannot have constraints on types for the latter;
- `tp_getset`, the table of getters and setters: it can be used, for example, to handle types not registered to Python, and to enforce types of `PyObject`s;
- `tp_methods`, the table of type methods;
- optional custom callbacks, to allocate the memory, create, initialize and finalize the objects etc.

This makes binding existing C++ classes possible; in doing that, there are different approaches to keep their instances in memory and manage their life cycle.

The first one is to have the instance entirely contained inside the Python custom object structure. In this way, the interpreter is always the owner of the instances, it decides when to allocate and to free the memory they use, and the programmer should use a *placement new* in the `tp_new` and `tp_init` callbacks, and explicitly call the destructor in the `tp_dealloc` one.

The advantage of this approach is that some class members can be registered in `tp_members`. A disadvantage is that there is no way to tell if the instance is valid, unless the data structure contains an additional flag to tell whether the constructor has been called. Another problem is that whenever an instance is

needed in Python, it needs to be constructed and managed by the interpreter, which might just not be a choice with some libraries.

An alternative is to store a pointer of the instance. This can be used to untie C++ instances from Python instances, allowing, for example, the library to be the owner of them, and to create different Python objects for the same C++ instance. The code of `tp_new` and `tp_init` would be easier, because there would an easy way to check if the object has been instanced, i.e. by checking if the pointer is `NULL`. However, in this case getters/setters become compulsory, and, more importantly, the memory ownership management is more difficult.

With C++ library that uses smart pointers like `std::shared_ptr`, there is also a hybrid approach: like the second one, method calls and member access are done through the pointer, but Python would own and manage an instance of the smart pointer.

Listing 11 contains an example code to expose a C++ class with the second approach; it shows that this operation requires a substantial number of lines of code, however also the other approaches would require a similar amount of lines of code.

5.3.3 Exposing functions

Exposing module functions and type methods is done in a similar way: they both require a callback with a prototype that matches the `PyCFunction` typedef, i.e. they should be in the format `PyObject *func_name(PyObject *self, PyObject *args)`.

The first difference between module and type methods is the array of `PyModuleDef` in which their information will be placed.

The second difference, is that type methods will likely have their specialized type for the `self` argument, e.g. `CustomType *self` instead of `PyObject *self`. This works without problems, but the callback pointer will need to be cast to `PyCFunction`.

The cast is needed also when the methods accepts *keyword arguments*, because the callback needs an additional `PyObject *` that holds the keyword arguments.

In general, the callbacks perform the following steps, as shown in Listing 12:

1. declare variables that will hold the *unpacked* arguments: `args` and `kw` contain arguments packed into a Python container, such as a tuple for arguments and a dictionary for the keyword arguments;

2. *unpack* the arguments, with `PyArg_ParseTuple` or `PyArg_ParseTupleAndKeywords`: these functions are like `scanf`, they take

a format string with the types the caller expects to receive, and a series of pointer of variables where to unpack these arguments; moreover these functions raise exceptions if the types in the format and in the packed arguments do not match;

3. process the arguments, if needed: check for validity of the domain, do some cast etc;

4. call the actual function;

5. cast the result to return it to the Python script.

5.4 Practical limits

As seen in section 5.2, using this API it would be possible to create the bindings of any C/C++ library. However, the amount of code generated is very high, as shown in listings in Appendix A. Doing that manually would take a long time even with a small quantity of classes to bind.

An automated tool would be helpful to create the bindings, but its output would be hard to maintain, because each class would likely require thousands of lines of code.

Chapter 6

Pybind11 and Binder

The code of Python extensions might become very long, and also very repetitive, e.g. the getters of properties change only by the cast of the type they return.

When using C++, templates can help in this respect: they could be used to create the callbacks to the functions, the getters/setters and also the custom objects that store the actual C++ instance.

This chapter is about Pybind11, a library that uses this approach, and about the other advantages it offers; finally we will briefly compare the solution we found with the more established one, based on SWIG.

6.1 Boost.Python and Pybind11

Boost.Python [1] is a library part of the Boost project and developed since 2002. Its objective is to reduce the code required for the creation of a type, of its members and methods, to something like a call to a function for each item to bind, taking as arguments, for example the pointer of the method to bind, and the name to associate to it in Python.

Another aim of the library is to do that in a non-intrusively way, i.e. without changing the C++ code and without having to manually create proxy functions. All the required information, such as argument and return types, are inferred using compile-time introspection.

They managed to reach their objective with template-based metaprogramming, and with a layer of libraries, based on C++ advanced techniques, that guarantee a wide compiler compatibility.

The C++11 version of the C++ standard made much of this layer unnecessary anymore: they improved the capabilities of templates, by adding *variadic templates*, i.e. templates with a variable number of arguments, and introduced concepts like *lambda functions*, *tuples* and *type traits*.

All these features, and RTTI (*RunTime Type Information*, available also in previous C++ standards) are used by another project, called Pybind11 [16], that provides a syntax and usage similar to Boost.Python, but without having additional dependencies, apart from the C Python API and the C++ STL (*Standard Template Library*).

Thanks to this, it is a *header only* library, thus it becomes part of the software

that uses it, but this is not a problem, because it is released under a permissive BSD-style license.

Using Pybind11 simplifies the work of the compiler and improves dramatically the build performance, with respect to Boost.Python: PyRosetta, the Python interface to the Rosetta molecular modeling suite, switch from the latter to the former, and noticed a compilation time improvement of 5.8 times, and a reduction of binary size by 5 times [20].

6.2 Using Pybind11

Pybind11 creates a sort of declarative language made of C++ instructions to create bindings.

```
class Example {
public:
  void a_method(int i, int j);
  double some_number;
  const int some_constant = 42;
}

PYBIND11_MODULE(example, m) {
  pybind11::class_<Example>(m, "Example", "The documentation of the Example class")
    .def(py::init<>())
    .def("a_method", &Example::a_method)
    .def_readwrite("some_number", &Example::some_number)
    .def_readonly("some_constant", &Example::some_constant);
}
```

Listing 1: The code to create a Python module with a new classs, using Pybind11.

Listing 1 shows that creating a new type with Pybind is easier and much shorter than using the C API, as shown on Listing 11.

Some of the steps described in section 5.3 are still visible, e.g. the module initialization function, hidden by the macro `PYBIND11_MODULE`, and the need for explicit Python names passed as strings. Some other steps, typically the low level mechanisms, are still present, but invisible and managed by Pybind11: the callbacks for methods, the getters and setters, the argument unpacking and check steps etc.

However, when needed, Pybind11 allows some manual control, about memory management, ownership, and other matters. Thanks to C++11 *type traits*, in particular `std::is_pointer` and `std::is_reference`, Pybind11 can understand if a function returns a pointer or a reference, and by default it takes the ownership. In almost all cases, we needed to keep the ownership, which can be

done simply by passing a flag as argument to `def`. Pybind11 provides also more advanced ownership models, such as object dependency, useful, for example, to create iterators on containers.

6.2.1 Callbacks

The example of Listing 1 is very simple, but also more expressive constructs from OOP can be exposed in an easy way, such as method overloads, operators and in particular class inheritance.

```
1  class Car {
2  public:
3    void accelerate();
4  }
5
6  class ConvertibleCar : public Car {
7  public:
8    void openHood();
9  };
10
11 PYBIND11_MODULE(example, m) {
12   pybind11::class_<Car>(m, "Car", "A basic car")
13     .def(py::init<>())
14     .def("accelerate", &Car::accelerate)
15     .def("brake", &Car::brake);
16   pybind11::class_<ConvertibleCar, Car /* this is the parent */ >(m, "ConvertibleCar", "A convertible
     ↪  car")
17     .def("open_hood", &Car::openHood);
18 }
```

Listing 2: The code to create a class and a children class.

This feature is very important because it is a way to implement callbacks in C++, used also in Inspire Studio.

Pybind11 allows that to call Python from C++ with classes that they call *trampolines*: they contain only the methods that can be overridden in Python, and the body of each of these C++ methods is intended to be just a call either to the macro `PYBIND11_OVERLOAD` or to `PYBIND11_OVERLOAD_PURE`.

However, we had to rewrite these macros, because they made Python callbacks take the ownership of objects returned by pointer, whereas we needed to make the Inspire Studio kernel keep their ownership.

6.2.2 Python wrappers

To implement our custom trampoline methods, we exploited the C++ wapper to the C API, part of Pybind11, in addition to the code for the extensions.

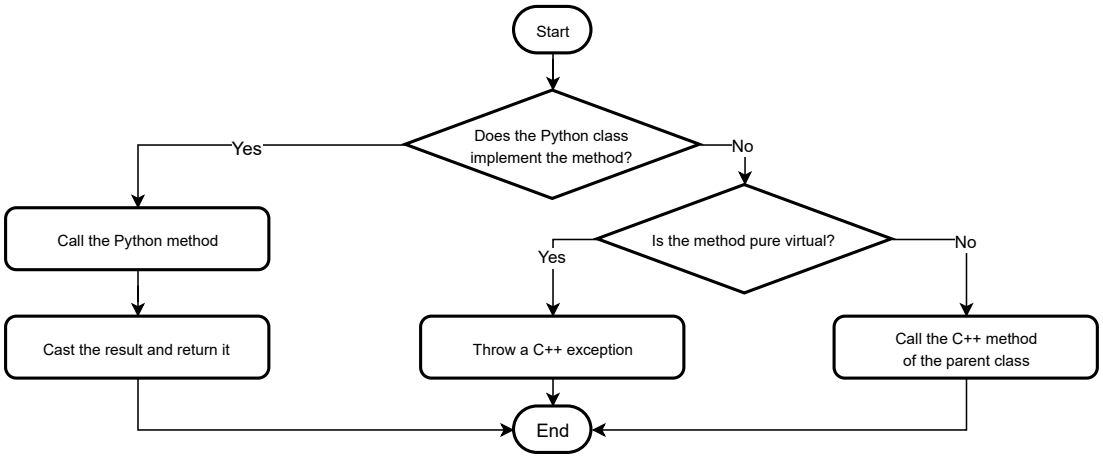


Figure 6.1: The flow of the method bodies of trampolines.

They provide a `pybind11::object` class, that encapsulates a `PyObject *`, performs automatic reference count with C++ constructors and destructors, and has some operator overloads, like `operator()`, which is used to call the callable objects (functions, types to instance them, etc), and which we used in the trampoline methods.

Pybind11 wraps also some classes from the standard Python library, such as `pybind11::list`, `pybind11::dict`, and also some abstract ones, like `pybind11::iterable`. They were useful to us because Pybind11 has a mechanism of implicit conversions, which we used to allow Python programmers to use Python native types instead of some classes normally used with the C++ PDK: to enable this behavior, we had to write some proxy methods, taking these C++-Python interfaces as arguments.

We used them also when we implemented some C++ functions to be used only by the bindings: if they needed a container, we made them accept any generic iterable with `pybind11::iterable`, instead of requiring a precise type, like `pybind11::list` or a container from the Inspire Studio utilities.

Finally, Pybind11 offers also some utilities to embed the Python interpreter, which allow, for example, to manage its life cycle as a normal member of a class. Moreover, by changing the `PYBIND11_MODULE` macro on the initialization function with the `PYBIND11_EMBEDDED_MODULE` macro, the bindings automatically become a built-in module.

6.3 Automating the bindings

Although Pybind11 has many advantages, it has also a considerable limit: it needs the developers to write all the code manually.

The authors of PyRosetta encountered the same problem, which they solved by writing a tool called *Binder* [19], based on *Clang libTooling*, "a library to support writing standalone tools based on Clang" [7].

Binder creates Pybind11 code by analyzing some header files first. It can be configured through command line arguments and through a configuration file, to make it generate the bindings for some classes, namespaces, functions, or to avoid them to be automatically flagged as to be bound. LibTooling based tools can also pass arguments directly to the Clang compiler, a feature that is useful, for example, to define macros, that in turn can be checked to enable or disable code sections.

To achieve its result, Binder performs three steps:

1. it invokes the Clang compiler to build the *abstract syntax tree*, or AST (Listing 13 is an example of that), which is then used to parse the desired C++ declarations and to automatically resolve and bind their dependencies;
2. it transforms the gathered data in C++ code;
3. it outputs some `.cpp` files, ready to be passed directly to the compiler.

The output files are easily understandable, which helps in finding the causes of errors, but the correct way to do definitive changes, to make the process repeatable without deleting modifications, is to use the configuration file, which offers also the possibility of calling custom binding code for classes and functions.

Binder is not a perfect tool, it has also some problems, but it is an open source project released under the permissive MIT license, therefore we could modify it to tune it to our needs. We will see more details about that, and about what we could achieve in chapter 7, when we will discuss about the results of the PDK bindings.

6.4 SWIG

Being born in 2016, Binder is a young project. Boost.Python had a similar approach, based on GCC XML, and on a Python script called *Pyste*, to convert its output to Boost.Python C++ code.

Another traditional way to create bindings is to use SWIG (*Simplified Wrapper and Interface Generator*) [2].

Established in 1996, this project has a wider aim, than the aforementioned ones. It focuses on automating the process, with its own C/C++ parser, and on targetting a high number of languages, including Python, C#, Java, Perl, Tcl/Tk, possibly with a few changes in inputs for each language.

For us, these features were disadvantages, rather than advantages. First, when we tried to use SWIG 3.0.12 (the version 4.0.0 had not been released, yet), we could not have its parser working on our files, in particular it had problems with templates. Indeed C++ is constantly evolving and becoming more complicated: using a major compiler as foundation might be just a stronger and more reliable option.

Secondly, a problem with having many target languages is that some decisions are taken in a way that they are good for all of them, instead of having many specialized options for every language.

As an example, not all of the languages supported by SWIG are object-oriented, but all of them have a C API, therefore it flattens the OOP structure to create an intermediary C version of the code to bind. With Python, by default this flattened code is bound in a library based module, then the class structure is recreated in a script module, adding more overhead; however this behavior can be disabled with the command line argument `-builtin`, to have better performances.

Thirdly, the output of SWIG is a single C++ file, ready to be compiled, and *intended not to be modified*: it is very long, and comprises a common boilerplate, the binding entirely made with the Python C API, and the customization passed as input, if any.

Modifying that file is difficult, and not recommended: there are other ways to do customization and avoid losing with subsequent runs of the process. However having readable files is important even just to understand what the bindings are doing, and debug them.

Microsoft presented the topic of C++ extensions for Python in the official documentation of Visual Studio: they recommend Pybind11 for C++ extensions and comment that SWIG is an "excessive overhead if Python is the only target" [25].

Chapter 7

The Python PDK

The information presented in chapters 5 and 6 is the result of a preparatory work we made to understand the Python extension world. During this stage, we also created some *toy models* to understand the feasibility and what we could expect from the project.

In this chapter, we will see how we passed from the preparatory stage to have the final bindings, the problems we encountered and how we enhanced the workflow.

7.1 Binding the B-Rep geometry library

As already mentioned in section 4.1, the PDK is divided in three parts: the first is interfaces to Inspire Studio, its UI and its features in general, the second is a B-Rep geometry library, and the third is a collection of utilities independent from Studio.

We decided to start from the second one, because it is smaller and simpler in several aspects:

- in an **absolute** sense: it is a normal, single, quite small dynamic library;
- in **dependencies**: this library has been projected to depend only on some utilities from the third part of the PDK, and on Parasolid, the geometry kernel used in Inspire Studio;
- in **variety** of programming techniques and patterns used: use of class inheritance is the most advanced technique the library use.

In particular, the first two points allowed us to try the bindings with a Parasolid sandbox application that we already had, before dealing with the additional problem of loading and starting the Python interpreter inside Inspire Studio.

Although we started without creating an advanced configuration, both Binder and the compiler succeeded without reporting any issue, but we could not import the compiled module in a Python script, because it failed in runtime.

The problem, simplified in Listing 3, was easy to resolve, but it was meaningful, because we did not expect the possibility of a success in compile time and a failure in runtime: it might be a concern in a more automated scenario.

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/embed.h>
namespace py = pybind11;

struct Parent {
  enum En {A, B};
};

struct Child : public Parent {
  enum En {C, D}; // This is completely unrelated to Parent::En
};

PYBIND11_EMBEDDED_MODULE(example, m) {
  py::class_<Parent> p(m, "Parent");
  py::enum_<Parent::En>(p, "En");
  py::class_<Child, Parent> c(m, "Child");
  py::enum_<Child::En>(c, "En"); // py::enum_<Child::En>(c, "ChildEn"); would work
}

int main()
{
  py::scoped_interpreter guard{};
  py::module::import("example"); // This will fail with "ImportError: generic_type: cannot initialize
  ↪  type "En": an object with that name is already defined"
}
```

Listing 3: A simplified example of the situation we had in the geometry library: we had two enums with the same name, one in a class and one in a child class. It compiles, as it is valid C++ code, but it does not run, because Python raises an `ImportError` exception in runtime. Renaming the enum, even just in Python, is a way to solve this problem.

In any case, after having solved also this issue, our test scripts worked flawlessly, and we decided to start the binding of the other parts.

7.2 Binding Inspire Studio interfaces

Our next step was trying to bind the classes that allow to create modeling tools and to manage their parameters. Since we already had the B-Rep geometry library, with the conclusion of this phase, we expected to be able to create some simple plugins that could actually do something in the Inspire Studio scene.

However, the creation of the bindings of this part of the PDK was more involved than the previous part, and we had to make changes in Binder, because its authors did not take some cases into account, or to make it more suitable to our requirements.

7.2.1 Opaque pointers and nested classes

When Binder finds and adds a function prototype or a class declaration to the list of candidates to bind, it checks also for dependencies:

- the types of the arguments and the return type, for functions and methods;
- the public member types and public parents, for classes;
- template arguments, for template classes, whereas template function invocations are ignored.

All these dependencies are added to another list, the one of needed types, unless they were disabled in the configuration.

The process is iterative: the first time, only the declarations that match the configuration are added to the candidate list, then dependencies are moved to the candidate list and the pass is repeated, starting from the candidates found so far. The loop exits when the candidate list stops growing.

We had a pair of problems with this mechanism. The first one is that Binder cannot handle correctly opaque pointers: they are created with forward declarations, that are added to the dependency list, but they will never be moved to the candidate list, because classes are added to the latter list only when their

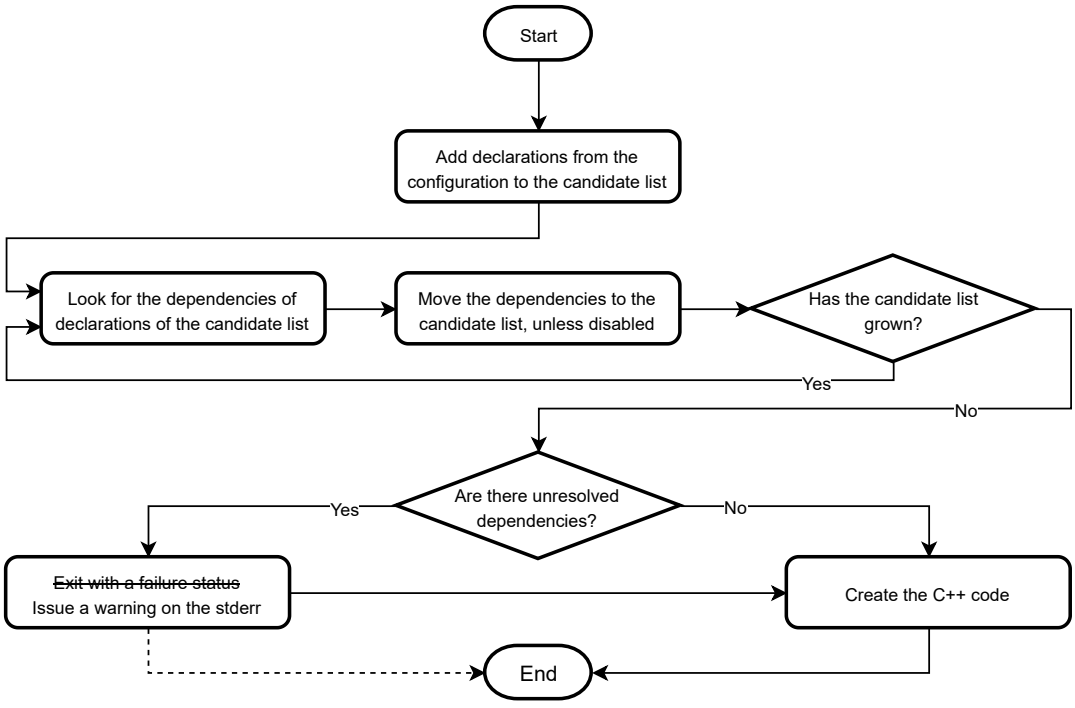


Figure 7.1: A simplification of how Binder works, without the interface with LibTooling. The original behavior ("exit with a failure status" and the dashed line) was a problem with us, so we changed it.

```cpp
#include <pybind11/pybind11.h>
namespace py = pybind11;

// These two structs can be everywhere in the codebase
struct Opaque;

struct Example {
  Opaque *ptr;
}

// This implementation can be made visible only to the bindings code
struct Opaque {
  void *placeholder;
};

PYBIND11_MODULE(example, m) {
  py::class_<Example>(m, "Example")
    .def_readwrite("something", &Example::ptr);
  py::class_<Opaque>(m, "Opaque");
}
```

Listing 4: This works and can be used without any problem. The opaque object will be created and managed within C++ code, but Python scripts will be able to pass it to functions or copy it to variables, if needed.

implementation is found. The behavior of Binder for these cases was to stop and call `exit` before creating the output.

In fact, Pybind11 can handle opaque pointers, with a code like the one in Listing 4; the opaque type still needs to be registered, however it is sufficient to pass a `class` or `struct` created just for the purpose to `pybind11::class_`, then everything works, because, although the content of the opaque class will be likely different, RTTI does not differentiate them. To be more precise, accordingly to our experience, for this to happen with the Microsoft Compiler, it is important to be consistent with the usage of keywords `struct` and `class`: a forward declared `class` cannot have a dummy `struct`, and vice versa.

Therefore we decided to modify Binder to issue a warning instead of an error, and let the process finish, so we can fix the problem manually if needed, action that does not always involve writing code.

Nested classes in the AST are children of the containing class (see Listing 13): when the latter is not bound, Binder does not visit neither its node, nor its children nodes, therefore it cannot find nested classes and treat them as forward declarations.

We could not obtain the bindings of a class that was a child of another class nested in a third class (see Figure 7.3), which we did not enable in the configuration. Initially, we could not figure out why this happened, but when we

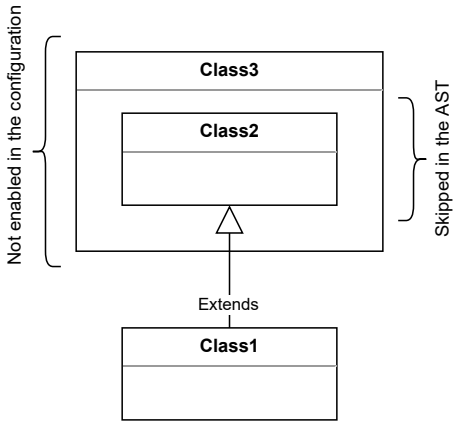


Figure 7.2: Nested classes cannot be bound, if their containing class is not enabled. Any child class will not be bound, as well.

understood the problem, we just added the containing class in the configuration file and everything worked as expected.

7.2.2 Templates

Thanks to the Clang infrastructure, Binder can deal with any template, but they need to be *instantiated*, which happens, for example, when they are members of a function, but not when they are used only in function arguments.

Indeed, we did not have some templates in bindings because they were used but never instantiated, so we made Binder issue warnings when this happens. We then rely on this information to create some code, specifically a dummy structure with them as members, with the only purpose of forcing their instantiation.

With this requirement met, template classes are bound as they were normal classes.

However, to make their name unique, Binder adds the argument values to it, but in this way it might become very long. `typedef`s are commonly used to solve this problem, but Binder does not export them. Since they are parsed and inserted in the AST, as shown in Listing 13, we added an handler for them, as we will see on Section 7.2.5.

Another problem with names was that Binder did not take into account that also pointers can be used as template arguments, therefore it did not have a filter for the `*`, which would be included in Python type names, although this is not legal, thus we had to add some code to replace the `*`.

7.2.3 Selection of trampolines

As already stated, the PDK uses many interfaces, in the OOP sense, i.e. the headers contain many abstract classes made only of pure virtual methods, and the actual realization is resolved in runtime, instead of being linked in compile time.

However, there is no way to understand if this is the logical function of an interface, or if it is made to make a callback structure available, or anything else.

The decision of Binder authors was to create the trampoline structure whenever it found an abstract class. Therefore, after the first time we ran Binder, we found a lot of trampolines in its output.

We decided to enable customization: we added the possibility of having trampolines enabled or disabled by default, and of having a granular configuration, i.e. enabling or disabling the trampolines for each class.

7.2.4 Enumerations and custom callbacks

In the rewriting from Evolve to Inspire Studio, several constants have been switched from C macros to both scoped and unscoped enumerations. While the former need to be explicitly passed with their type, the latter can be cast to integral types.

Some functions of the PDK that logically accept constant from unscoped enumerations, have `int` in their prototypes, but Binder cannot mark the correct `enum`s automatically as dependencies, in this way. Thus, we added also the possibility to specify `enum`s to bind in the configuration, at least as a temporary solution to use while the prototypes have not been fixed.

Some other constants are still implemented as macros, therefore we added them using a custom C++ function that takes the module `PyObject`, and we added the possibility to specify the name of callbacks to call from the module initialization function in the configuration.

7.2.5 `typedefs`

Some parts of the PDK use templates, especially in the utilities, and in some cases, some templates are even nested, resulting in names longer than 40 characters and no mnemonic at all. In many cases, if not all, commonly used template specializations in the PDK have `typedef`s to make them meaningful and their use easier, so we decided to automate a similar mechanism also in Python.

Python does not actually have `typedef`s, because it does not need them: as discussed in Section 5.2, types are also objects, and, as objects, they are attributes

```python
class a_very_long_class_name:
  pass
alias = a_very_long_class_name
print(alias, type(a_very_long_class_name()), type(alias()), isinstance(alias(),
↪  a_very_long_class_name)) # <class '__main__.a_very_long_class_name'> <class
↪  '__main__.a_very_long_class_name'> <class '__main__.a_very_long_class_name'> True
```

```cpp
class SomeCppClass {};
PYBIND11_MODULE(example, m) {
  pybind11::class_<SomeCppClass>(m, "a_very_long_class_name");
  m.attr("alias") = m.attr("a_very_long_class_name");
}
```

Listing 5: The way to create `typedef`s in Python, and the equivalent version with Pybind11. The Python example shows also that the `a_very_long_class_name` class is an attribute of the module `__main__`.

of something else, in general of a module or of another class, and they can be copied to another variable.

Therefore we added to Binder some code to do the same in C++: take the context where the `typedef` is, create a new attribute there, named as the `typedef` identifier, and assign the original type to it.

Listing 13 shows how to create a type alias in Python, and how to do the same in C++ with Pybind11.

To be sure that all the types have been registered when this alias creation is run, we added it at the end of the module initialization function. However, in this context we do not have the objects of the created types, thus we need to traverse the whole hierarchy of types and of where the aliases will be, including nested classes, if necessary.

7.2.6 Changes in the output of Binder

We were not satisfied with the structure of the output created by Binder: they were divided by input header file, and then the bindings of each header were split so that the size of each output file would not exceed a certain threshold. In this way, finding a certain class was hard.

We decided to have a file for each class, instead. This solved the unpredictability, made files easier to manage, and also sped up the building process in certain cases, e.g. when only one class had been modified. In some other cases, e.g. a clean build, it slowed down the process, because some common code is built in each compiled unit.

C++ names are case sensitive, whereas file are not, in Windows, thus, since

we had some classes that had the same name with different cases, eventually we grouped them in the same file.

Another characteristic that we did not like, is that Binder always put the explicit cast of the function pointer in the `def` calls. In fact, it is needed only with methods that have overloads, which most often is not our case. Since it makes the files quite longer and harder to read, we decided to leave the casts only when strictly needed.

Finally, we modified Binder to let Pybind11 handle default values for function arguments, instead of creating a lambda with less parameters, and leaving the task to embed default values to the C++ preprocessor. The original strategy had the advantage that when default values changed, it was only necessary to recompile the bindings, instead of running Binder again. The disadvantages were that it made the code longer, harder to understand and that it did not fully exploit Python potentialities, shown in Listing 6.

Getting from the AST the default value of an argument is not a trivial task: it involves traversing and correctly combining an arbitrary number of nodes, i.e. all the descendants of a `ParamVarDecl` node. We handled several cases, e.g. numeric constants, unary and binary operations, parentheses, other kind of combinations, but we could not handle some cases, for which we left the original behavior.

```
void method(int a = 5, char b = 'x', int c = 7, int d = 42);
// To change only c, a anb b also needs to be specified:
method(5, 'x', 15);
```

```
def method(a=5, b='x', c=7, d=42):
  # Do something
# Can specify only the value of c, using it like a kwarg:
method(c=15)
```

Listing 6: Difference in how C++ (the listing at top) and Python (the listing at bottom) treat default arguments.

7.2.7 Manual bindings

Even with these customization, eventually we had to implement some binding code manually.

Sometimes it was because of language differences, hard to address in an automatic way. It is the case of functions that use pointers as arrays, or the functions that use reference parameters to return multiple values. In Python we used generic `iterable`s and multiple packed return values, respectively.

In other cases, we wrote manual code to make Python developers feel writing code with the PDK more natural.

7.3 The Python Interpreter in Inspire Studio

When the code of the bindings was ready, we had to face the problem of integrating the Python interpreter in Inspire Studio. In particular, we absolutely wanted to avoid the interpreter to be finalized when Python code was still in use or referenced, because this would crash the program.

The best way to solve this problem would be to do it in the core of the application, but we preferred using the C++ plugin architecture, both to satisfy the requirement of keeping the bindings separate, at least at the beginning, as described in Section 4.2.1, and to have more agility in testing.

Indeed, the basics of C++ plugins are that they are DLLs loaded into the program, and that they must implement a class to register their components. Although it is not endorsed as a feature, an instance of that class is kept alive as singleton and destructed only when Inspire Studio exits. For test purposes, we decided to keep a `pybind11::scoped_interpreter` as member of this class, and to use the same dynamic library also to keep the bindings, that are loaded as built-in modules.

This choice had also another advantage: we used the C++ plugin also to register components (modeling tools, importers and exporters) written in Python, and to instance them, when needed.

In particular, we wrote a Python modeling tool that adds a file dialog to choose any Python script, and a button to run it.

At this stage of the project, we also encountered some problems with the building process.

The first one is that all the metaprogramming structure used by Pybind11 makes the process very slow: a clean build in release mode took almost 11 minutes. On the same computer, a clean build of Inspire Studio 2019.3 in release mode takes slightly more than 22 minutes: adding the bindings to the main build would increase the time to complete it by 50%.

Templated code can become quite large also in size: we had to increase the number of sections in the `.obj` files with the `/bigobj` switch in Visual Studio [21]. We even needed to change the 32bit toolchain, used by default by Visual Studio also to build 64bit executables and libraries, with the native 64bit one, otherwise the linker would have crashed. In release mode, with many optimizations enabled, it needed more than 5GB of RAM at peak.

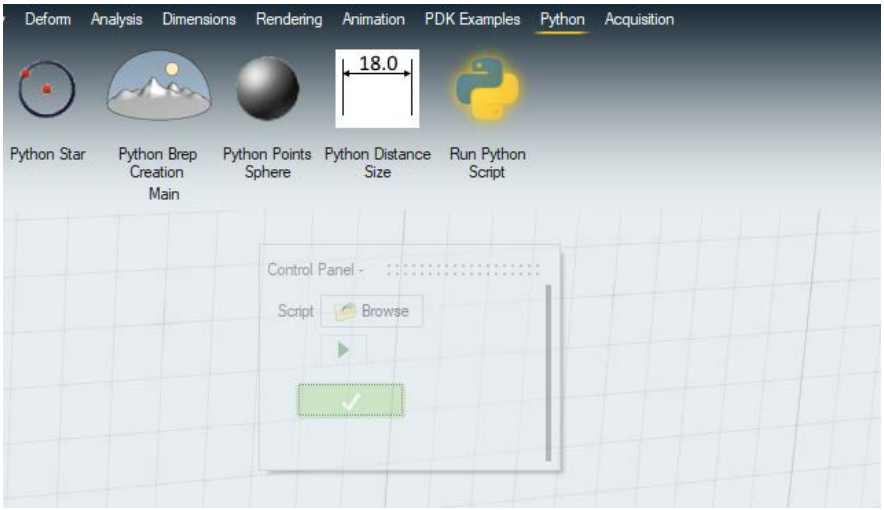


Figure 7.3: One of the tools we wrote: it allows to load and run any Python script. The tool itself is written in Python, thanks to the `importlib` module.

Another problem is that sometimes understanding the causes of building errors is hard, because Visual Studio does not report all the substitutions it has performed to arrive from the templates to the final code that is compiled. In some cases, even with such list, it would be difficult to understand the whole process, because the chain of replacements might become very long, and Pybind11 uses very advanced features of C++. We used only Microsoft Visual Studio 2015; we do not know how more recent versions of the MSVC compiler and Clang treat errors.

We encountered many linking problems due to the way symbols are exported from DLLs with `__declspec(dllexport)`, especially with implicit constructors, destructors and assignment operators, which we often resolved by expliciting them or by marking them as deleted.

In other cases, the bindings triggered the compilation of templated code that had never been used before, uncovering errors that had never appeared.

Both situations were solved by modifying Inspire Studio code. However, this required a new revision of the PDK, and the temporarily removal of some features from the bindings.

7.4 Updating the bindings

The PDK is extracted from the Inspire Studio codebase, therefore each build of the software has its own PDK, both identified with an identical version number. For the bindings to work with a certain version, they need to be updated and rebuilt for it.

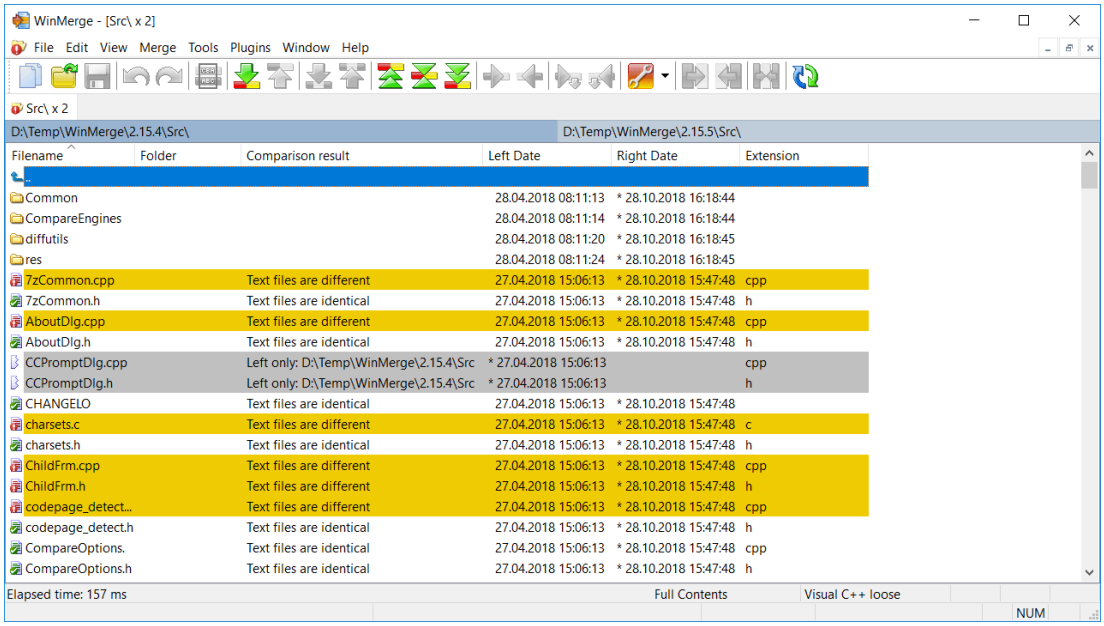


Figure 7.4: We use a specialized software to check the differences between the old bindings and the new output; in particular we chose to use WinMerge. This screenshot comes from the application website.

In our workflow, this is done by running Binder again on the updated headers, and then integrating the changes of its output, using a specialized software to compare the new files with the ones already in the project directory.

With this in mind, we tried to make Binder output as repeatable as possible, by having one class for file, as discussed in Section 7.2.6, and also by removing the declaration line number from the comments that Binder automatically adds to its output, because they very likely change, from a version to another.

This does not reduce the compilation time, because many units are built again even when they have not been modified, if some of the headers they depend on have. The real reason we did this, is to track the changes more easily, and to be able to identify the origin of new errors, if any, or to check if some modifications to the configuration or some manual bindings are needed.

We updated the bindings several times, since their first version, and with this workflow, everything worked almost flawlessly. Indeed, we observed that having a solid Binder configuration is helpful.

7.5 The example tools

Before starting the bindings development, we created some C++ examples on how to use the PDK, because the API of Inspire Studio is very different from the one of SolidThinking Evolve, therefore old samples were not useful anymore.

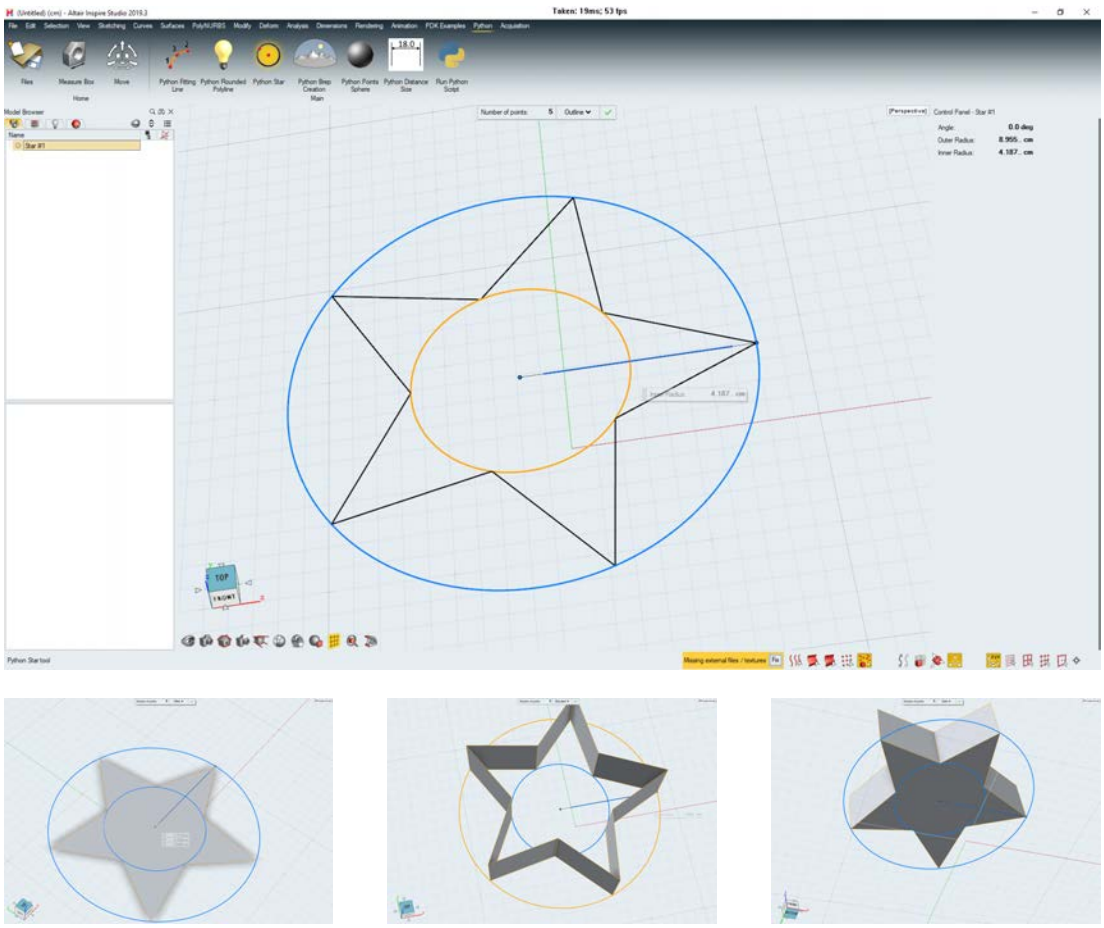


Figure 7.5: This simple modeling tool is an example about the various UI elements that parameters can be associated to: the *guide bar* at top, the *modeling tool panel* on the right, graphical *handles* and *micro dialogs* on the scene. It also shows that the same body can be created as a *wire*, *sheet*, or extruded, with our without top/bottom faces, thanks to the geometric library.

The examples show how to create plugins, modeling tools, importers and exporters, how to interact with the 3D scene, create and update objects, how to interact with the rest of the UI, how to create parameters and handle their changes, etc.

Once the bindings were ready, we tried to rewrite the examples in Python, as a first serious test to understand what the bindings were capable of. Eventually, the logic of the tools was not changed, and the work consisted in adapting the code from C++ constructs to Python ones. Many parts were refactored just with the *search and replace* tool of the text editor.

The only exception to this was the *Brep Creation* tool: in C++, it needed many big `switch`es, whereas the Python version could exploit the dynamic creation of function arguments in runtime, and the dynamic typing applied to functions with different prototypes, to create a table with a row for each possible

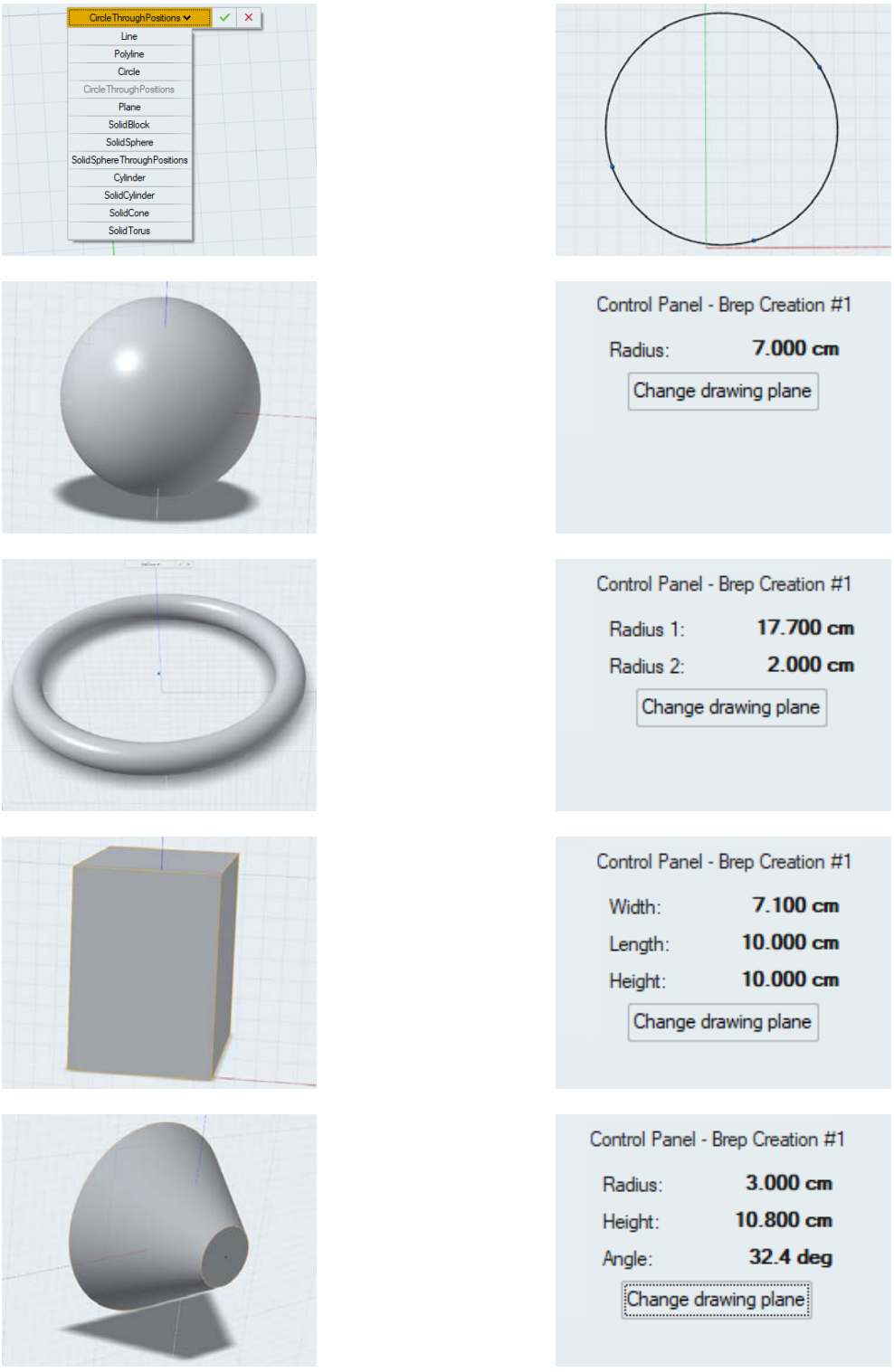


Figure 7.6: *Brep Creation* allows to create a lot of different shapes, but it need to keep the UI updated accordingly.
Doing this in Python was easier than in C++ and Python code for the tool is much more compact than the C++ code for the equivalent one.
In the figure, there are shapes with the respective parameters, except for the circle through 3 points, since the points are moved directly in the 3D scene.

```python
# The real creation functions, similar to these ones, are actually part of the PDK module, they are listed here just to show their signatures
def create_square(edge_length):
  # ...
def create_circle(radius):
  # ...
def create_rectangle(width, height):
  # ...
def create_rhombus(edge_length, angle):
  # ...


class Shape:
  def __init__(name, callback, need_height=False, need_angle=False):
    # Save the params

shapes = [
  Shape('Square', create_square),
  Shape('Circle', create_circle),
  Shape('Rectangle', create_rectangle, True),
  Shape('Rhombus', create_rhombus, need_angle=True),
  ]
```

Listing 7: *B-Rep Creation* manages output shapes in a similar way. The Shape objects are used to handle several features of the modeling tool, such as the output creation, the shape list selector, etc. Doing something like that in C++ is more involved and requires templates.

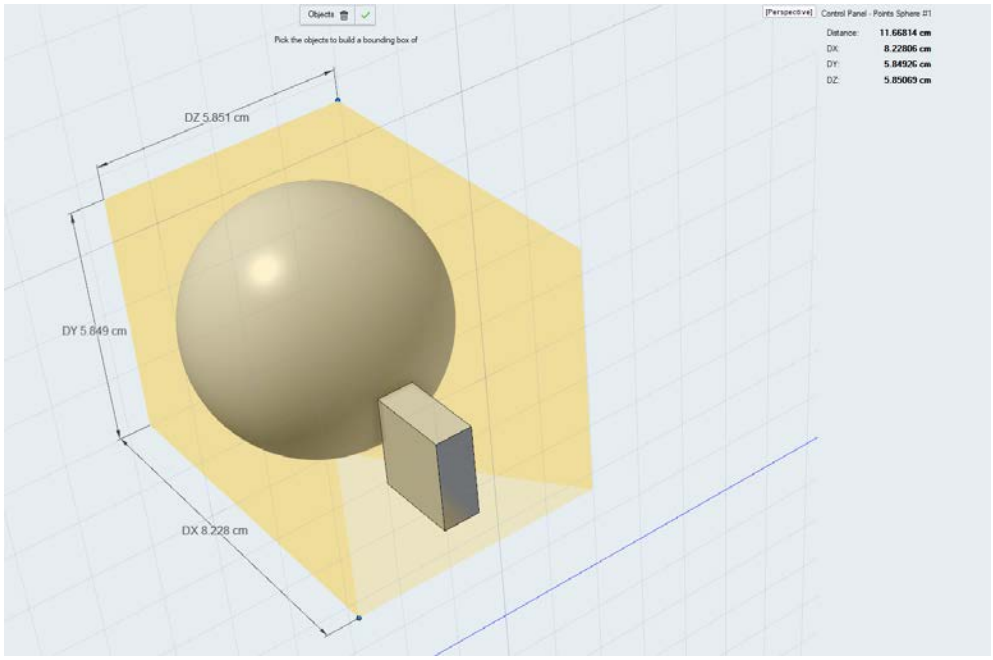


Figure 7.7: Like C++, Python allows also to create tools that are just helpers and do not create or modify anything, like this one, that make the user pick some objects, and then shows a bounding box around them and its sizes.

output shape.

To load the Python modeling tools, we exploited the same plugin that embeds the interpreter, we have not implemented a more generic loader, yet.

From an operating point of view, C++ example tools and Python example tools are indistinguishable: they work in the same way and have the same features.

7.6 Future developments

Inspire Studio is available both for Microsoft Windows and for Apple macOS, and the PDK is meant to be a platform agnostic way to create plugins for both the operating systems, with the same code.

We tested the Python bindings only in Windows, however we expect the bindings to work also in macOS with no modifications at all, or with a few changes, but even in this case, we think that we would be able to find an automatic way keep do them in further revisions.

A build on macOS should be one of the first steps to do before continuing with other developments.

After that, we should make the core of Inspire Studio aware of the Python support: we could keep the integration in a dynamic library, but instead of being a plugin, it should be internal, and have the proper calls for initialization and finalization. This is already planned, as is the Python console and another way of loading Python plugins.

Then we should do further tests, also to extend the coverage of features that are known to work.

Documentation is another important problem: at the moment, the C++ PDK is documented with Doxygen. An idea is to make it include also the Python documentation, as the OpenCV project does, since the Python API and the C++ one are almost identical. Another approach would be to use Sphinx, the tool that many major Python projects use; when we tried to use it, we did not manage to make it load all the binary dependencies of the bindings.

It would be also helpful to have a feedback from PDK user, to understand what are its weaknesses. In case, we could try to make the API more comfortable to Python developers.

About Binder, our changes include many trials and errors, therefore our modifications are not ready to become a pull request for them. However in future we might try to formalize them and request to have them merged.

Part III: The Acquisition Plugin

Chapter 8

Acquisition of 3D models

A way to design 3D models is to start from a blank scene and draw everything from scratch, or by taking inspiration from 2D images, if needed.

Another approach is to start from existing 3D models, or even from physical objects.

There are many reasons to do that:

- building a component that fits with an existing product, such as a replacement or a compatible new part;
- improving or changing the design of existing products;
- recreating legacy projects that have been lost;
- creating parts that would be very difficult, or impossible, to create from scratch, like some organic shapes.

Different steps may be involved in the process:

1. **scan** of the model: this stage needs some specialized device, like a 3D scanner or a depth camera, its output is usually a *point cloud*, i.e. a set of points that do not have connectivity information;
2. **clean** of the data: in this step, the acquired data is cleaned, and, if needed, transformed in some other geometric representation, e.g. a triangle mesh;
3. **parameterization**: one might be interested in a mathematical model of the object, e.g. a NURBS based one, rather than a polygonal model, however this step is optional, because sometimes a polygonal model is enough.

Our plugin allows to do a scan from an existing model with a RGBD camera, then it creates a triangle mesh which includes also color information, but it does not create a NURBS model from that.

Chapter 9

RGBD sensors

RGBD sensors are cameras that provide at least a color stream (RGB), and a depth stream (D), although some have also other features, e.g. infrared capture and audio acquisition.

Typically, all this data requires an interface with an enough high throughput, such as USB 3. Sometimes a gray stream is preferred to colors, because it needs a lower bitrate, or in some other cases only the depth stream is enabled, since it is enough for reconstructing 3D models as point clouds.

In this chapter, we will see three types of depth sensors, then we will show how to use depth data to reconstruct a 3D scene, and finally we will introduce the Intel RealSense D415 Depth Camera, the device we support in our acquisition plugin.

9.1 Depth acquisition techniques

Traditionally, 3D have been done with LiDAR, *Light Detection and Ranging*. Thanks to lasers, it can be very precise, its error is in the same order of magnitude as the light wavelength, but LiDAR scanners are also expensive.

During years, alternative techniques to measure depth with other cameras have been developed, for use cases that need less precision and that can benefit from affordability. Some examples are *stereo depth*, *time-of-flight*, and *structured light*.

A common limit in all of these approaches is that very reflective objects alter the measurements. For this reason, it is important also to choose adequate backgrounds: matte ones are preferred, especially if they are highly textured, because they interfere less with the acquisition.

9.1.1 Stereo depth

This technique is very similar to what human eyes do. It employs two cameras, whose distance, or *baseline*, is known, and allows to compute the depth using the correspondences from one image to the other. More precisely, the depth is inversely proportional to the difference in distance of corresponding image points and their camera centers [32].

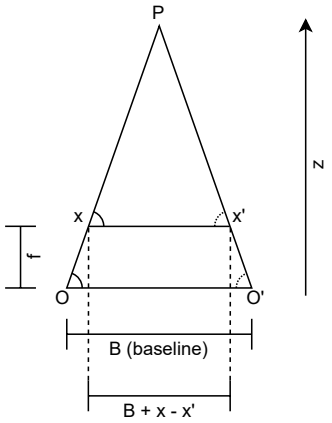


Figure 9.1: The simplified case of parallel cameras. x and x' are the distances between points in image plane corresponding to P and their camera center.

Figure 9.1 represents the simplified case in which cameras have parallel optical axes, and the corresponding points are only translated [13]; the generic case is similar, but it should take also rotations into account.

We can exploit the similarity between the two triangles (same angle in P, and two corresponding angles in parallel lines), from which it follows:

$$\frac{z}{B} = \frac{z-f}{B+x-x'} = \frac{z-f}{B+d} \tag{9.1}$$

where d is the *disparity*, that is defined as $d = x - x'$. With some computations, we obtain

$$z = \frac{Bf}{d}. \tag{9.2}$$

This technique can work with any pair of normal cameras, indeed OpenCV has a module [32] for stereo vision.

However, specialized hardware, like the Intel RealSense Depth Camera D400-Series, include ASICs (*Application Specific Integrated Circuit*) to find correspondences in hardware, and perform depth acquisition in realtime, up to 30FPS.

Moreover, the D415, D435 and other specialized cameras work on infrared wavelengths, to allow adding some texture with a noise projector, which can improve the measurements, without influencing the color data.

The advantages of stereo vision are that it is more robust to ambient illumination, therefore it can work also outside, with direct sunlight, and that multiple devices can operate together, without any interference.

Some disadvantages are that it needs two sensors that need to be calibrated together, and that repetitive structures can make the determination of correspondences ambiguous. Computational complexity, due to correlation algorithms, is another disadvantage, although ASICs are a solution to this problem.

9.1.2 Time-of-flight

Time-of-flight, or TOF, involves measuring the time that a wave, or a particle, takes to reach a target and come back. With a model of the motion equation, it is possible to compute the length of the traveled path from these duration values.

TOF is a generic term: this method works with any kind of wave, e.g. acoustic or electromagnetic, but, usually, 3D scans employ IR sensors.

The advantages of TOF depth sensors are that transforming the received signal to a depth measure involves just some simple computations, and that sensors are very compact, indeed phone producers have started to include them in their flagship smartphones.

A first disadvantage is that they are not normal cameras, but specialized chips. Another problem is that they are influenced by background illumination, therefore many of them perform poorly outside, and multiple sensors may disturb each other. They also suffer from the indirect paths problem, i.e. they may provide incorrect results because some rays might be reflected more than once.

The second generation of the Microsoft Kinect relies on a TOF sensor.

9.1.3 Structured light

The structured light approach requires a projector and a camera. The former projects a known pattern on the scene, whereas the latter, since it has another point of view, detects deformation of the pattern made by the surfaces: by analyzing them, the system can compute the depth.

Varying the pattern during time allows the capture of more details. Also, changing parameters of the pattern may produce different results.

Structured light shares many of the disadvantages of TOF: interference from ambient background light and multi-device interference. They also suffer from multi-path effects, but in a lesser extent, with respect to time-of-flight.

However, as an advantage, the structured light can achieve a higher precision, especially when merging several frames.

The first generation of the Microsoft Kinect is a structured light camera.

9.2 3D reconstruction from depth

For reconstruction purposes, depth is the most important stream: by knowing its intrinsic parameters, i.e. the focal length and the principal point position, and by knowing the units of the sensor (e.g. 1 unit is 1mm), the depth data alone

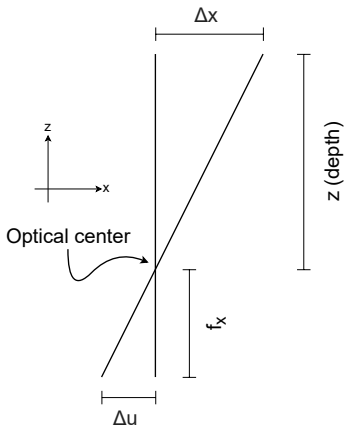


Figure 9.2: A simplified model of a pinhole camera, that does not take distortions into account, with the XY plane being the focal plane, and with the optical center in the origin, projected on the XZ plane. Δx is in real world units, as z, whereas Δu and f_x are measured in pixels. On the YZ plane, the situation is similar. Refer to [12] for a complete and detailed discussion of the camera model.

is sufficient to obtain a 3D representation of what the sensor is looking at, as a monochromatic point cloud.

With Figure 9.2 as a reference, Δx can be found exploiting the triangle similarity: we have two right triangles (by construction), with equal angles at the principal point (vertically opposite angles). Therefore,

$$\frac{\Delta x}{z} = \frac{\Delta u}{f_x} \Rightarrow \Delta x = z\frac{\Delta u}{f_x} \tag{9.3}$$

and, in a similar way, we obtain

$$\Delta y = z\frac{\Delta v}{f_y}. \tag{9.4}$$

By doing these computations for each pixel of a depth frame, it is possible to build a point cloud. With fixed intrinsic parameters, $\frac{\Delta u}{f_x}$ and $\frac{\Delta v}{f_y}$ are constant terms for each pixel. Therefore, it is possible to build a matrix with these values, and exploit array programming to improve performances when elaborating multiple frames, e.g. for realtime rendering of what the sensor sees.

Also, a simple form of triangle mesh can be obtained by linking points that comes from adjacent pixels, if their distance is below a certain threshold, although, as we will see, to completely reconstruct an object, it is necessary to integrate several frames. This is needed to have data also on occluded parts of the object, and to reduce the error; some algorithms have been designed for this purpose.

The color data can be applied to the point cloud, to make it more realistic, but it can be also required for reconstruction purposes, in addition to the depth data, e.g. in RGBD odometry algorithms.

In any case, being physically different sensors, color and depth images have different extrinsic parameters, and in some cases they have also different sizes, aspect ratio and field of view. *Registration* is the post processing step that allows to overlap them; usually, it is done in the host via software, by calling some function provided by the sensor SDK.

9.3 Intel RealSense D415

We wanted to offer the possibility of interfacing a depth camera to scan objects directly in Altair Inspire Studio.

Our first requirement was to support a device that was still in production, so that if the plugin will be released to public, interested people will be able to obtain a device to use it. This excluded the Microsoft Kinect 2 immediately.

Another requirement was that the cost of the device should be adequate to what it offers.

Eventually, we chose the Intel RealSense D415, because of its technical features and also because it is very compact, therefore also maneuverable.

Figure 9.3: The Intel RealSense D415 Depth Camera. Photo by Intel.

It provides color data at 1920×1080 pixels, and depth data at a maximum resolution of 1280×720 pixels at 30FPS. The depth is computed with stereo vision in the IR, and the device includes an IR projector as well, that can be enabled or disabled.

An advantage of the solution by Intel is that it provides also some useful features in the SDK, such as a filter to preserve edges, or a temporal filter to improve the depth data persistence.

Another benefit is that Intel provides official support also for macOS. Although it is not as complete as Windows and Linux ones, the stream capabilities work as intended.

9.3.1 D415 Limits

The D415 allows the depth scale to be set either to 1mm or to 100µm. Since the value of each pixel is a 16bit unsigned integer, the theoretical maximum depths are 65.535m, or 6.5535m, depending on the depth scale. However there are also other factors that limit the bounds of what can be measured [11].

The first one is that the *Intel RealSense Vision Processor D4*, the ASIC of the D415, can look for correspondences in a window of 126 pixels. This window can be moved, with the parameter that the SDK calls *disparity shift*: the default window is from 0 to 126, with a disparity shift of 10px, the window goes from 10 to 136.

Given a certain disparity value d (e.g. 126, or 10, or 136), by knowing the baseline B (55mm, for the D415), the width w of a frame (e.g. 1280px) and the horizontal field of view F (65°, for the D415), the corresponding depth z can be computed with the following formula:

$$z(d) = \frac{fB}{d},\ f = \frac{w}{2\tan\left(\frac{F}{2}\right)}. \tag{9.5}$$

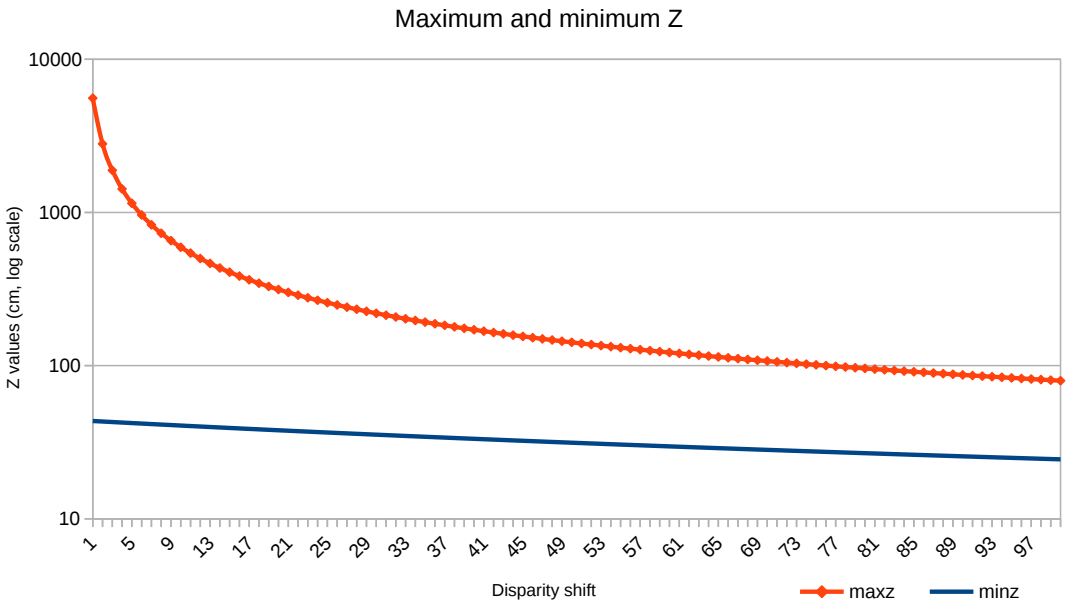
Equation 9.5 connects any disparity value to a z value, and it makes possible to compute the minimum and the maximum depths that the RealSense can measure. In particular, by default, at the maximum resolution, the minimum z is about 44cm, which can be far for some applications. One way to reduce it is to decrease the stream resolution, but this increases the error on z. Another way is to change the disparity shift: this reduces also the maximum depth, which might be not a problem, considering that many applications already have a threshold, and that with the default value, theoretically the maximum depth is unlimited (we would have 0 at the denominator, in Equation 9.5, but, as we will see, in practice there are also other limiting factors).

As Figure 9.4 shows, setting disparity shift to a value like 41px might be a good tradeoff, because it allows to scan everything in the 33cm-134cm range, but setting it to a value like 95px to arrive to 25cm of minimum depth reduces the maximum depth to 58cm, that might be too low, to comfortably move around an object.

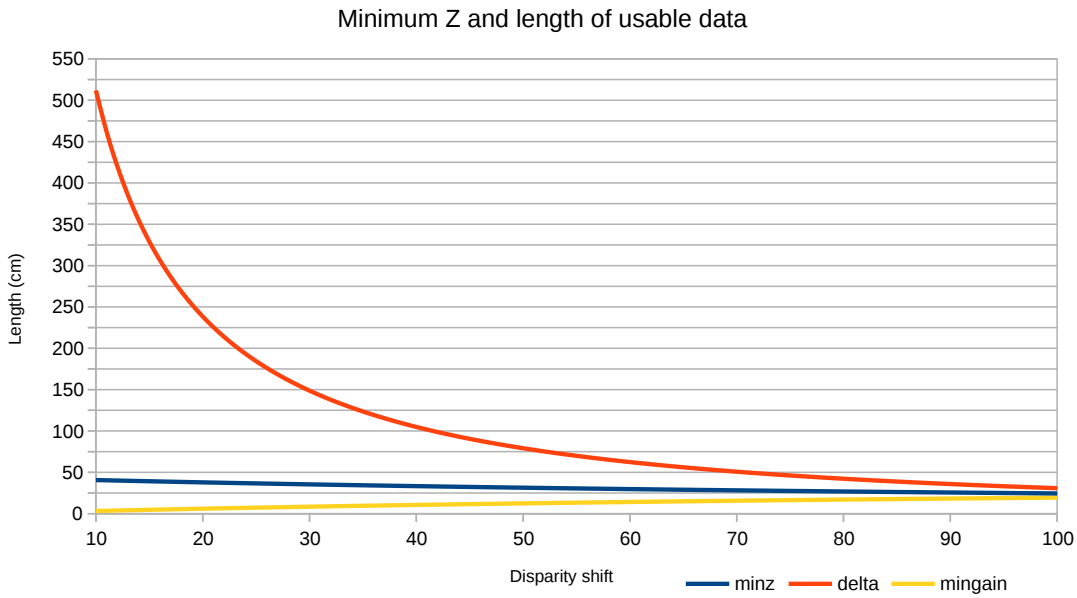
An additional reason to stay closer to the object to scan is that the RMS error on depth grows with the square of z:

$$e = \frac{sz^2}{fB} \tag{9.6}$$

where s is the subpixel RMS error, which is "virtually independent of distance to the target" [14], and can be measured with a specific software.



(a) Maximum and minimum depth in function of the disparity shift. The scale is logarithmic to include both the values in the same graph.



(b) The length of the usable area is in red, and the additional minimum depth, with respect to disparity shift = 0 is in yellow. To gain a few centimeters of minimum depth, it is necessary to lose a lot of usable range.

Figure 9.4: Plots of the Equation 9.5, for the parameters of the D415 and 1280px of width.

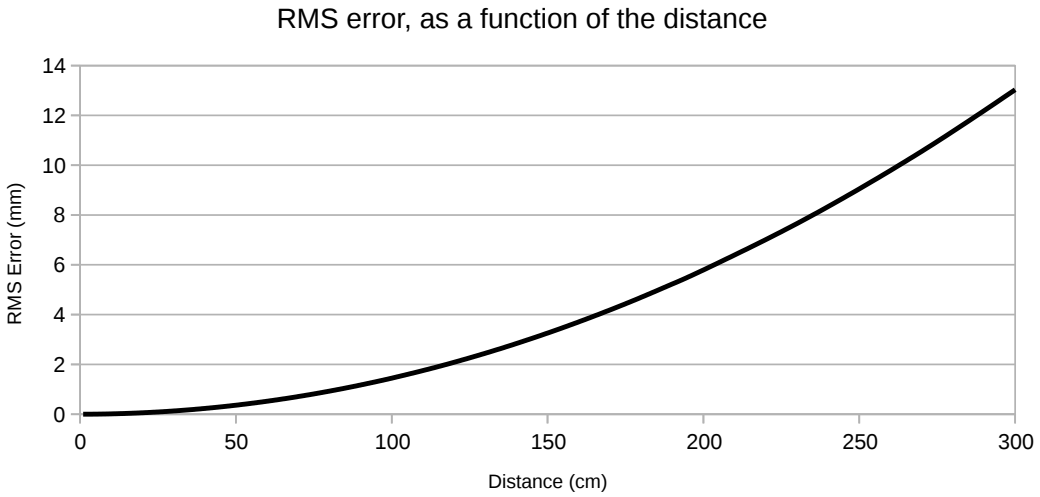


Figure 9.5: RMS error, with the same parameters as above, and 0.08 as subpixel RMS error

Distance is not the only factor that introduces errors, but there are also many others, e.g. wrong exposition or interference of reflective materials.

Indeed, the D400 series has many parameters that can be tuned to enhance the results, but some of them are not documented, because they are intended to be set through the presets offered by the SDK, that are optimized for a certain purpose, using machine learning [6].

Chapter 10

The workflow

During years, the problem of interfacing with real world scenes, objects and environments, for many different purposes, has been studied: several workflows and techniques have been developed and improved. Some of the proposals published in literature have also available software implementations.

We chose *Kinect Fusion* [23] [15]: as its name states, it was designed for the Microsoft Kinect, but it needs only depth frames and the implicit parameters of the camera, data that any depth sensor can provide. In this chapter, we will briefly see how this pipeline works.

10.1 Kinect Fusion

Actually, Kinect Fusion is based on previous techniques: its authors started from ICP (*Iterative Closest Point*) [4] and TSDF (*Truncated Signed Distance Function*) [8], which they adapted to exploit the parallel computation power offered by GPUs, to process subsequent depth frames in realtime.

The reason is that the objective of Kinect Fusion is to build a complete reconstruction while acquiring the data, without further processing.

To achieve this, the movements of the camera should be small and smooth, because ICP, as we will see, needs the point of view of each frame to be close to the previous one. A fast algorithm is helpful, because longer times correspond to larger movements.

Like all the TSDF-based methods, Kinect Fusion represents the acquired scene as a voxelized point cloud. In a first setup step, an empty volume is created, then an iterative and incremental update procedure, that takes only the depth frame as input, performs the following steps, for each frame:

1. **depth map conversion**: the frame is preprocessed to reduce noise and converted to a point cloud;

2. **camera tracking**: with ICP, a rigid transformation, i.e. rotation and translation, is computed and then applied to the point cloud, to match it with the rest of the volume;

3. **volumetric integration**: the point cloud is voxelixed and integrated in the existing volume;

4. **raycasting**: the new volume is rendered from the last camera position, to create a synthetic depth map, passed to ICP on the next frame.

10.1.1 Depth map conversion

The first steps are preprocessing of the frame: first a threshold of maximum depth is applied, which, accordingly to our experience, can speed up the next phases. Then a bilateral filter is applied, to reduce the noise while preserving the discontinuities. At this point, each pixel is transformed to a point, as described in Section 9.2.

In the parallel implementation, each compute unit calculates the coordinate of a single pixel, and its normal vector from its neighbor points, needed by the *point-to-plane* optimization of ICP [30].

10.1.2 ICP

Iterative Closest Point is an algorithm for scene registration.

Given two point clouds $\mathbf{P}$ and $\mathbf{Q}$, that are **already roughly aligned**, it looks for correspondences $\mathcal{K} = \{(\mathbf{p}, \mathbf{q})\}$, and it builds a linear system from them to find a transformation matrix $\mathbf{T}$.

Then $\mathbf{T}$ is applied to points of $\mathbf{Q}$, and the process is repeated, to find an improved matrix. The exit conditions are given by a target energy function $E(\mathbf{T})$, that the algorithm minimizes, and by an iteration counter.

In the original ICP implementation, called also *point-to-point* ICP, this energy function is:

$$E(\mathbf{T}) = \sum_{(\mathbf{p},\mathbf{q})\in\mathcal{K}} \|\mathbf{p} - \mathbf{T}\mathbf{q}\|^2 \tag{10.1}$$

whereas, in the point-to-plane ICP it is:

$$E(\mathbf{T}) = \sum_{(\mathbf{p},\mathbf{q})\in\mathcal{K}} ((\mathbf{p} - \mathbf{T}\mathbf{q}) \cdot \mathbf{n}_\mathbf{p})^2 \tag{10.2}$$

and it is the only difference between the two methods.

Thanks to, $\mathbf{p}$ and $\mathbf{q}$ are detected as correspondences if $\|\mathbf{p} - \mathbf{T}\mathbf{q}\| \leq d$, where d is a threshold. Several workflows perform registration with methods that exploit feature vectors, or 2D algorithms like SIFT or ORB, or similar methods, and then they run ICP as a refinement.

Kinect Fusion uses only ICP, which simplifies the operations, because otherwise at each update, new feature vectors should be computed, but it is also a strong assumption. This is the main reason for the camera movements to be small and smooth.

Kinect Fusion was designed to run *online*, therefore, if the camera tracking fails, the user should be warned and they should go back to a working pose.

10.1.3 TSDF

The volumetric integration is made on a uniformly sampled grid of voxels, whose size and pitch can be chosen accordingly to the application.

Each voxel has two properties associated: a truncated signed distance, averaged through the various frames, and a weight, which is the times a voxel appeared in a frame.

For Kinect Fusion, the signed distance is the difference between the distances of point and of surface to camera. A positive SDF means that the vertex is in front of the surface, whereas it is negative if the vertex is behind the surface, thus zero crossings define the surface.

Only the pixels whose SDF is within a certain threshold are considered in the update, and, for this reason, this method is also called *Truncated Signed Distance Function*.

The advantages of this kind of representation are that it encodes uncertainty in the data, it allows to merge multiple measurements easily, and to fill holes thanks to them. Moreover, with the distances it is possible to do interpolation, to place points or vertices with subvoxel precision, when extracting a point cloud or a triangle mesh.

The parallel implementation employs a computing unit for each (x, y), which sweeps along the z-axis.

To improve speed, the grid is stored as a linear array, which guarantees higher performances than hierarchical structures, in GPUs, even though the latter are much more memory efficient. A voxel needs at least a 32-bit float (16-bit *half precision* floats might be sufficient, however many consumer grade GPUs do not support them) and a 16-bit integer for the weights. Depending on resolution, the grid can occupy from a few MB, e.g. 100 elements in each direction, to some GB, with more than 1000 elements in each direction. In fact, memory is the main limit to volume space and resolution.

10.1.4 Raycasting

Running ICP between two subsequent frames might produce bad results, as sometimes frames have a high ratio of missing data due to indecision of the sensor, or errors due to artifacts. Therefore, at the end, the update procedure creates a target cloud for the next update: this synthetic data is less noisy, and more

globally consistent.

With this information, the procedure creates also a 2D rendering of the surface, that can be displayed to the user, to show whether the camera tracking is working correctly.

This operation runs in parallel, as well: each computing unit cast a ray and finds a single vertex/render pixel.

10.2 Marching cubes

Kinect Fusion does not have any finalization step: after any volume integration, the reconstructed data that it can offer is ready.

However, besides being memory inefficient, generally a voxel grid like that one is not supported by 3D modeling software.

A first approach is to convert it to a point cloud, i.e. take only the voxels on the border of the surface, interpolate their coordinates using distances, and put them in a point list.

However, the marching cubes algorithm [18] can work directly on that kind of data and generate a triangle mesh, that can be used with almost all software. To keep the same terminology of its introductory article, 0 is our "user-specified value", and the signed distance is the "vertex data-value", in this way it finds the zero crossings.

This algorithm is very simple and can be implemented in a very efficient way without parallel computations.

A disadvantage of marching cubes is that it does not simplify the generated mesh in any way: its vertex will be sampled like the original grid, and an additional simplification step might be desirable in the pipeline, to save memory and because a simpler model often implies better performances.

10.3 Other approaches

TSDF based algorithms, like Kinect Fusion, understand which regions of space are empty, thanks to visibility information, but many other approaches to surface reconstruction [3].

Some assume that the scanned scene meets some geometric properties, such as surface and volume smoothness, and exploit them to provide a reconstruction. Others try to reconstruct the scene as a set of simple geometric shapes, like cubes, spheres etc. Another category is the one of *data driven* algorithms, that try to

reconstruct single objects, by composing them as parts of existing models, or by using them to complete missing parts in the scan.

Chapter 11

Implementation of the plugin

We decided to implement the scan functionality with a single plugin, that offers three modeling tools:

1. **acquisition**: this tool interacts with the RealSense, saves the captured frames and in the meantime runs also Kinect Fusion, to create an initial reconstruction. It also warns the user in case the camera tracking fails;

2. **refinement**: this is an optional stage that runs again Kinect Fusion on the already acquired data to obtain a more precise tracking;

3. **mesh extraction**: this modeling tool converts the point cloud obtained with Kinect Fusion to a standard triangle mesh.

In the rest of the chapter, we will see how we implemented them, the problems we encountered and how we solved them.

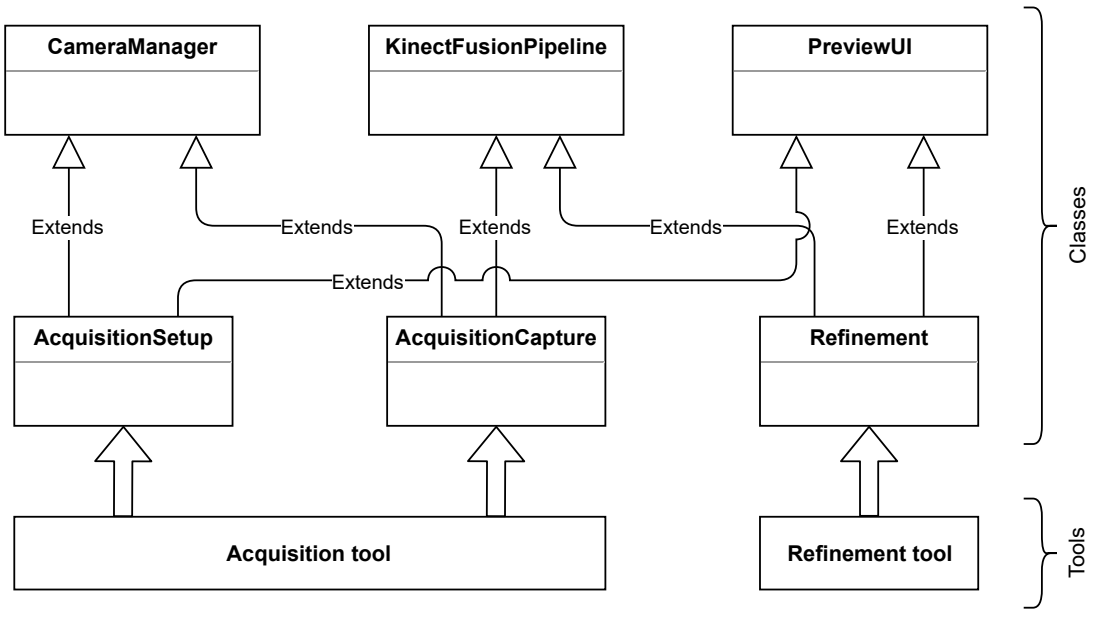


Figure 11.1: The structure of the acquisition and of the refinement classes and tools. As we will see, they are very similar, and share several portions of code.

11.1 Acquisition tool

This tool is the most complex of the three, and the one that has to provide most of the functionalities.

First, the users choose parameters such as the object dimension and position, then they start the actual scan, that provides a first reconstruction of the object, as a point cloud.

11.1.1 Setup

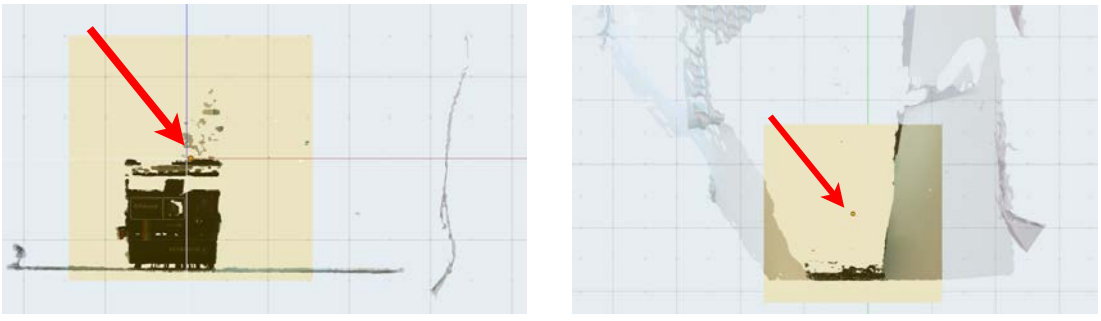
In this phase, the users set the various parameters, and a preview of what the sensor sees is displayed in the scene, as a triangle mesh, created with the simple algorithm explained in Section 9.2.

To make the target area more visible and more distinguishable, we show the volume bounding box and we apply some transparency to the parts that are outside it. Users can move the target region by dragging its center with the mouse in this preview, or by inserting its coordinates in the modeling tool panel.

Then, users are asked for a path where they want the tool to save the parameters file and the frames.

We chose to save the parameters in a JSON file, which contains the volume properties, and the information about the dataset, i.e. the intrinsic parameters of the camera, the depth scale, and the transformation matrices, once they are available.

For color and depth frames we chose the JPEG and the PNG formats, respectively; both will have a numeric suffix, and files with the same numbers, e.g. `color_00001.jpg` and `depth_00001.png`, are to be considered a single aligned



(a) Front view (b) Top view

Figure 11.2: The preview of what the RealSense is framing. The plugins supposes that the user is looking at the front, indeed, the top view does not have much data. The small orange dot is the handle to move the cube.

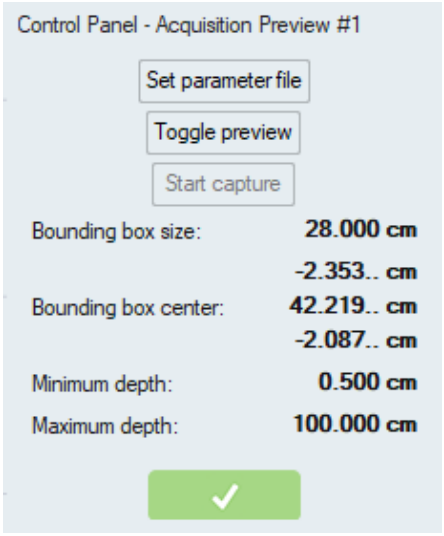


Figure 11.3: The parameters that can be set on the preview. Notice that the "Start capture" button is gray because the file has not been set yet.

```
{
  "depthScale": 9.999999747378752e-05,
  "failed": [1093, 1094, 1095, 1096, 1098, 1099, 1100],
  "initialPose": [
    1.0, 0.0, 0.0, -0.16353767363786697,
    0.0, 1.0, 0.0, -0.16087585596046448,
    0.0, 0.0, 1.0, 0.2821944129562378,
    0.0, 0.0, 0.0, 1.0],
  "intrinsic": [
    928.3314208984375, 0.0, 638.6371459960938,
    0.0, 927.9815673828125, 352.825927734375,
    0.0, 0.0, 1.0],
  "maxDepth": 10000.0,
  "minDepth": 0.0,
  "online": true,
  "size": [1280, 720],
  "transforms": [[
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0],
    null],
  "volSize": 0.30000001192092896
}
```

Listing 8: An example of parameters file. Notice that the transformation of the second frame is `null` but it is not in the `failed` array: it means that it has been skipped, which may happen, since they are the result of the realtime tracking, as the `online` attributes shows. The file contained also the other transformation matrices, even though they have not been shown here.

frame.

While the volume parameters can be left to defaults, if they make sense, the output directory is compulsory, and the button to start the recording remains disabled, until the user provides it.

11.1.2 Capture

The capture receives frames from the RealSense, saves them, calls the Kinect Fusion update, and if the camera tracking fails, it warns the user.

Apart from the interface with the sensor, we decided to do these tasks with the OpenCV library: image input/output is one of its basic functionalities, and it provides an implementation of Kinect Fusion (`cv::kinfu`, [24]), already optimized to exploit GPUs via OpenCL.

During the capture, we also update a point cloud on the Inspire Studio scene. However, we noticed that it is not the best way to show if the capture is working correctly, because neither Studio, nor OpenCV support colored clouds.

Therefore we decided to display the color stream, with the depth stream overlapped as a heatmap, for which we call `cv::addWeighted` and `cv::applyColorMap`, respectively.

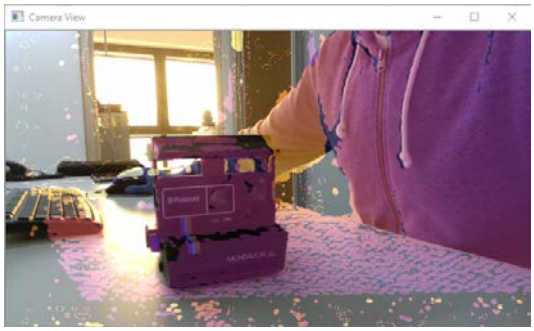
In another window we display the Kinect Fusion render, that helps to understand whether the tracking is working, because its point of view should be the same as the one detected for the camera.

Both the RealSense SDK and OpenCV offer Python bindings: we tried to write the tool in Python, but eventually we had to do that in C++. The first problem was that, as shown on Listing 9, in the Python version, the intrinsic camera parameters of KinFu are a read-only matrix, and without changing them, it is impossible to run the algorithm.

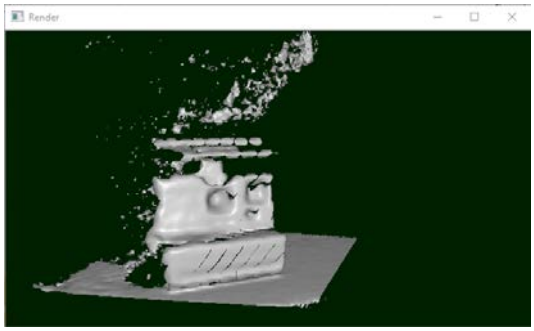
Although we managed to modify the source code and make it possible to set

```
1 struct CV_EXPORTS_W Params {
2 // ...
3     CV_PROP_RW Size frameSize;
4     /** @brief camera intrinsics */
5     CV_PROP Matx33f intr;
6 // ...
7 };
```

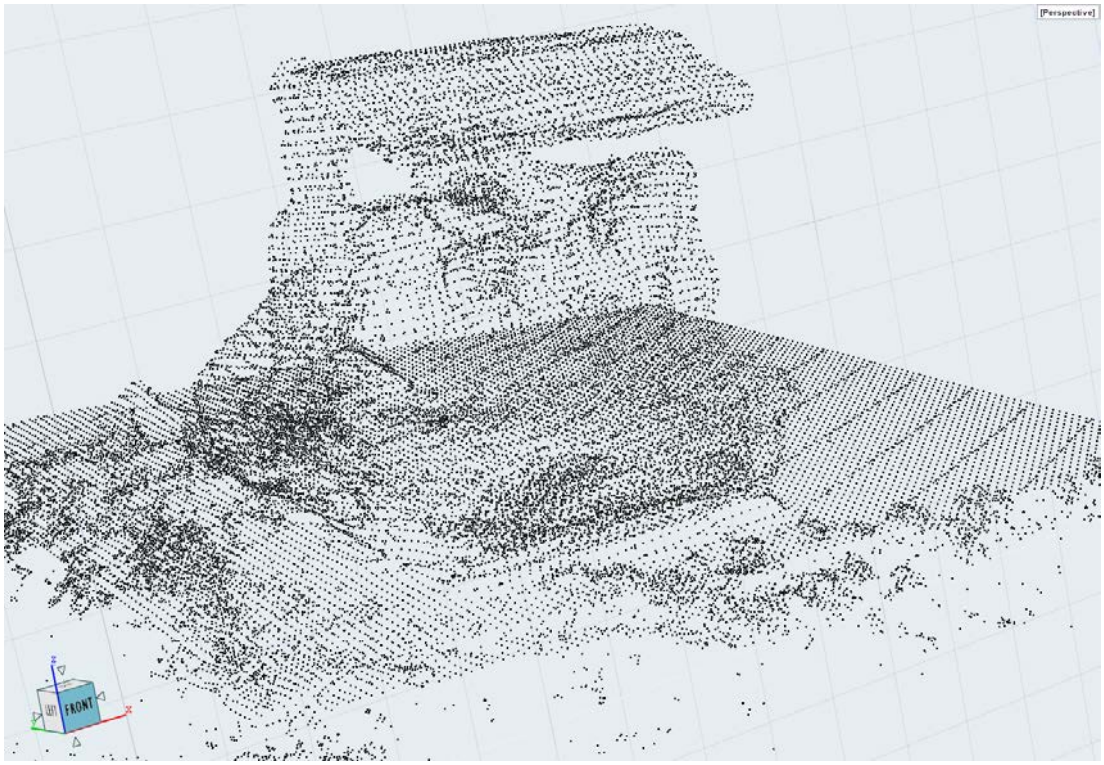
Listing 9: This is an extract of the struct that contains KinFu parameters. OpenCV automate the bindings with a tag based system. This listing shows that `frameSize` is marked as a read and write property, whereas the intrinsic matrix is just read-only.



(a) Color and depth, blended together.



(b) The partial render of Kinect Fusion.



(c) How a point clouds is shown by Inspire Studio.

Figure 11.4: We display two other windows, in addition to the main Inspire Studio one, for the acquisition, to make it more clear for the user whether the sensor is able to compute depth information and whether the camera tracking is working correctly. They are easier to understand, than the point cloud.

this matrix from Python, in some cases our scripts crashed because the KinFu update function threw some `std::runtime_exception`, that were not converted to Python exceptions, and this issue proved to be critical.

But the most problematic aspect is that the capture needs at least three threads: one to interface with the camera, one for the Kinect Fusion update (although it runs on the GPU, the call provided by OpenCV is blocking) and one to save images. When we tried to do this in Python, we encountered many

problems, both with the `threading` and the `multiprocessing` modules. The former had fairness issues, whereas the latter had problems with the inter process communication.

Indeed, for plugins with purposes like this one, i.e. hardware interfacing, and realtime processing, C++ remains a better option.

Even with native code and with the OpenCL optimization enabled, our test machine, which had a 6 year-old Nvidia Quadro K3100M, was not enough fast to elaborate all the frames and the tracking always failed.

Thus, for this stage, we decided to save as many frames as possible, and to check if the camera tracking is working by setting coarser parameters to ICP, and a bigger voxel size, rather than attempting a perfect reconstruction while capturing.

Therefore, we employ a simple lock to synchronize the acquisition and the volume update, because it needs just the last frame and the critical section is limited. In detail, the frame is kept in memory as a `cv::Mat`, and when it is instantiated with the copy constructor, the new object will share the same buffer of the source, rather than having its own. Thus, in the critical section, we just copy a pointer and increase an integer, used as reference count.

Instead, the file output thread needs all the frames, therefore, we synchronize it with the acquisition with an open source, lock-free queue, based on atomic operations, released under the BSD license [9].

Saving frames is slow, and we noticed that it continues also for some seconds after having stopped the acquisition. However, rather than the additional time, that may be considered as an "acqusition finalization", memory usage might be problematic: frames that are waiting to be written are kept in RAM, but in our tests they never exceeded 200MB-300MB, which is acceptable.

11.2 Refinement tool

As a second, optional step, users can run a tool to execute again Kinect Fusion on an existing dataset. Since it does not have to interact with the camera, it is not time constrained, and it can run ICP with parameters tuned to obtain more precise transformation matrices.

Moreover, the refinement tool is device independent, so any existing dataset that can be converted to our format, can be reconstructed with our plugin, which is possible also because this tool does not use existing transformation matrices.

We designed it this way to allow users to change the volume properties, if needed, which is useful to reconstruct objects from datasets that had wrong vol-

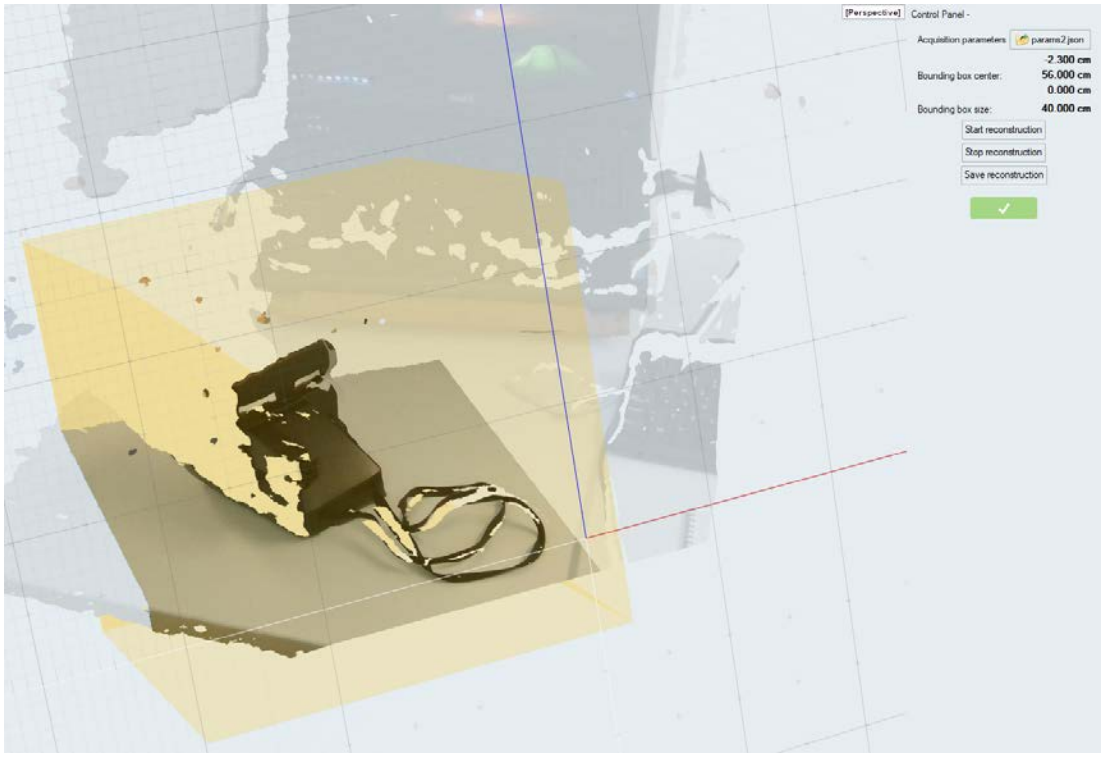


Figure 11.5: The refinement tool is very similar to the capture one. Indeed both have the possibility of setting the center of the volume and its size, but the former shows the first frame on the preview, whereas the latter shows a live preview.

ume position or size, without having to capture them again.

From a technical point of view, this tool is very similar to the acquisition tool, therefore we decided to share all the code that we could, and it resulted in a structure similar to the one represented in Figure 11.5.

11.3 Mesh extraction tool

The final step of our pipeline is the mesh extraction: from the TSDF volume, we extract a triangle mesh.

Initially, for this tool, we encountered two problems.

The first was that in the OpenCV implementation the TSDF data is not accessible: one can obtain only a point cloud, as list of vertex coordinates and normals. The `KinFu` class is abstract, and the actual implementations are not available in the headers.

The second problem was that OpenCV does not manage vertex color data.

To resolve both the problems, we copied the OpenCL kernel of the integration, and we modified it to handle also colors: we assign to each vertex the average of

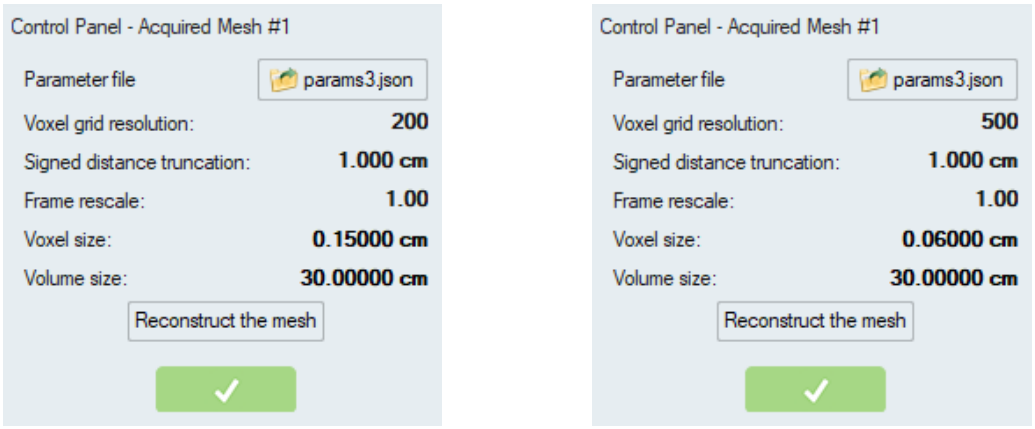


Figure 11.6: The parameters of the mesh creation tool. Volume size is fixed from the previous tools, and the voxel size depends on the resolution, as the two different screenshots show.

the colors which that vertex had in each frame in which it appeared.

Since we handle the modified kernel, we also have to manage its memory, and so we can access the TSDF data. In addition to this memory space, we pass the matrices computed with the first or the second tool.

For marching cubes, we modified an implementation based on lookup tables [5], to make it directly use the signed distance stored in our grid as surface value.

In our implementation, we added an index buffer and we made it reuse vertices, if they have already been checked and added, so that the output mesh is connected, which is important for many purposes, including retopology algorithms.

As an optional step, we allow to simplify the mesh. For this, we found a Python module version of an open source project called *Instant Meshes* [17]. Users can specify a certain target number of triangles they want in their output, and this algorithm reparametrizes the mesh.

Moreover, they can run this step many times, until they are satisfied, without the need to run the extraction from the grid again, because the original model is always kept in memory, therefore this operation is quite fast.

Since we did not have the same performance requirements like the ones of the acquisition tool, we wrote this tool in Python.

In the next chapter, we will show some reconstructed models, both before and after the simplification step.

Chapter 12

Results

To test the plugin, we tried to scan some objects.

In this chapter, first we will see some generic considerations about the reconstruction parameters and about the limits of the workflow. Then we will see the reconstructions of a couple of datasets. Finally, we will compare our implementation decisions, with the ones of Microsoft 3D Scan [22], a free software available in the Windows 10 Store.

12.1 Parameters

We noticed that the most important factor to obtain a good scan is to keep the scan volume as big as the object. E.g. if the object fits in a cube of 25cm on each size, it is better set the volume size to 30cm on each dimension, rather than $1 \times 1 \times 1$m.

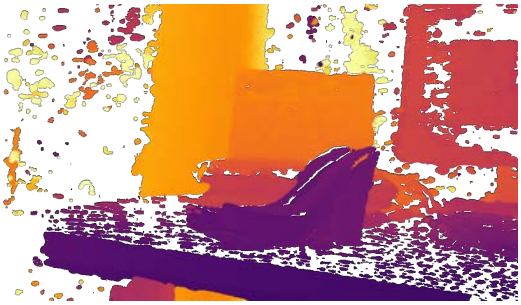
This is helpful for two reasons: first, with smaller volumes, it is possible to keep a high precision with a smaller grid resolution, thus faster computation times. Secondly, we observed that smaller volumes produce better reconstructions: we think that, in this way, ICP is forced to look for correspondences mainly in the target object.

The RealSense preset was another crucial setting: we switched from *high accuracy* to *high density*, contrarily to what the official documentation suggests [6]. Accordingly to our experience, the implementation of Kinect Fusion from OpenCV works with the latter, whereas, with the former, it almost always lost the camera tracking at a certain point, in our experiments.

However, one of the first datasets we captured with this preset had issues: the target object was duplicated with an offset, as shown in Figure 12.1, but we are not sure that the error was strictly connected to the preset.

As Figure 11.6 shows, we exposed also two other parameters: the truncation value for the signed distance and the resize factor for the frames. The former depends on the object that is being reconstructed: a lower value produces a model with sharper edges, but also with more holes, whereas a higher value corresponds to more rounded edges and to more filled holes.

The resize factor influences mostly the time of reconstruction, rather than the results, accordingly to our experience. We set the stream size to 1280×720px,



(a) The heatmap of the depth.

(b) The corresponding RGB frame.

Figure 12.1: In the depth frame, the shoe model is double, whereas the color frame is just a bit underexposed. We do not know if the problem was caused the preset or by another error with the device: the previous 15 depth frames of the same dataset were like the camera was still, whereas the corresponding color frames show movement.

and we observed that we could use 0.5 as size factor without any problems. Since each side is resized, with 0.5 as factor, the amount of data is reduced by 4 times, therefore the mesh extraction is faster.

12.2 Problems of the plugin

We managed to produce quite good scans of the objects, but we observed some problems in all the datasets.

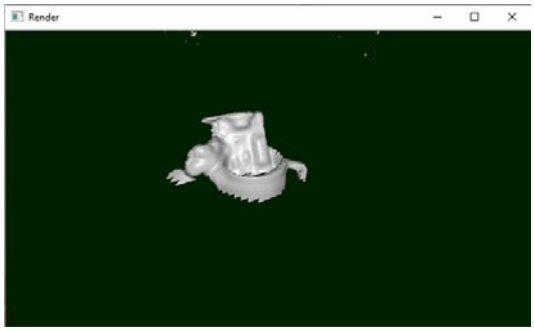
The first one is that sharp edges tend to become round, dependently on the distance truncation value. We also tried to apply a median and a bilateral filter, but we could not solve this problem.

Another characteristic of all the datasets is that small details, that are available in a single frame or in a restricted number of frames, disappear in the final reconstruction, even when the voxel resolution is enough small to include them. We think that a reason might be that in the final reconstruction they are treated as errors, maybe because they are not aligned correctly, since the transformation matrices likely have errors. Also, the averaging procedure implicitly smooths the model.

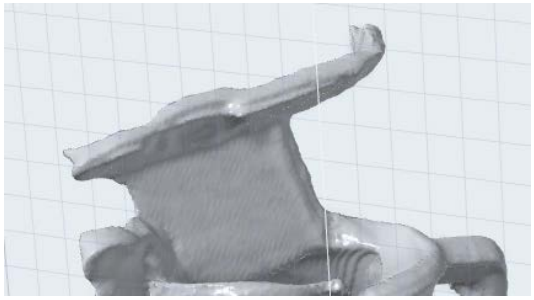
This is visible especially in Figure 12.2: the object that was being scanned was an electronic board, and the render of Kinect Fusion shows the integrated circuit and some other electronic components that were on it, whereas the final reconstruction does not.

Sometimes, reducing the voxel size helps to include more details.

We also encountered a technical problem with the visualization of the models



(a) The partial render of Kinect Fusion.



(b) The final reconstruction.

Figure 12.2: These two pictures represent the detail of a scene, that included an electronic board. On the left, the partial reconstruction made with raycasting is very detailed, it is possible to see, for example an integrated circuit. On the right, with the reconstruction process, these details have been lost.

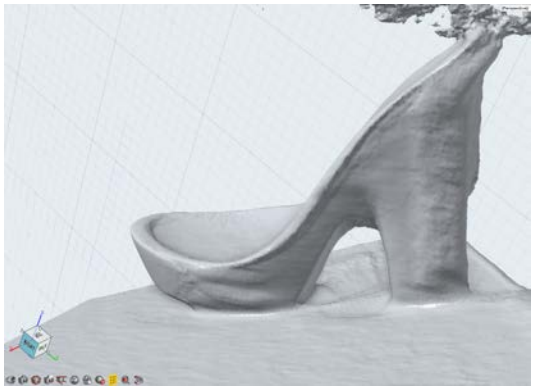
with their color. In the scene mode, we could not apply lightning to the visualization with materials, therefore it becomes slightly difficult to understand correctly the model, and in many cases the default material is clearer.

12.3 Datasets

With the first dataset, we tried to reconstruct a wooden model of shoe, used by artisans. The second dataset is about the instant camera that was the objective also of screenshots in Chapter 11.

All these pictures show that the reconstruction resembles the real objects in a close manner, but it is quite difficult to obtain a clean scan without any kind of noise.

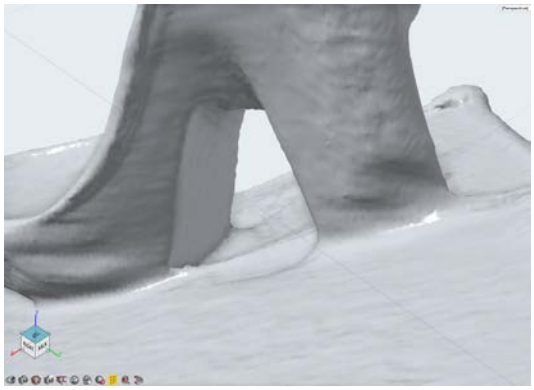
Moreover, the objects cannot be placed directly in a broader scene, or 3D-printed, but they likely need to be cleaned manually, first.



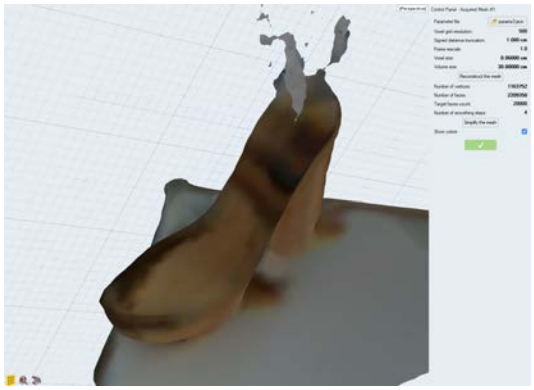
(a) A side of the model.

(b) The other side of the model.

(c) The front of the model.



(d) A detail of the sole.

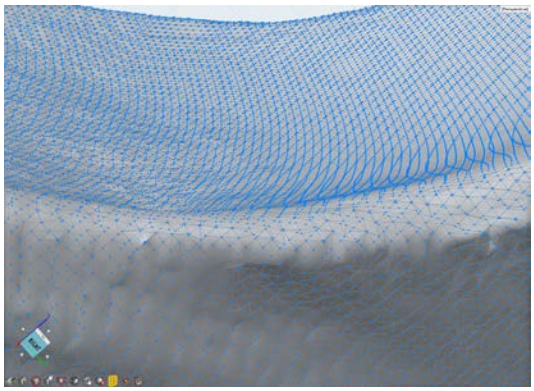


(e) View with colors.

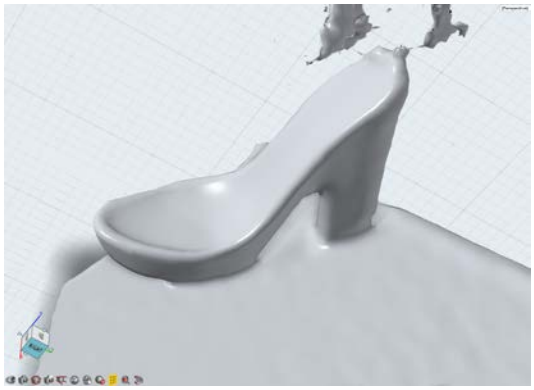
(f) One of the RGB frames.

Figure 12.2: The reconstruction of this other model. Again, there are some problems with the sharp edges. There is also some noise on the back of the shoe, due to the non optimal background of the acquisition environment, but it is disjoint from the model, therefore it is easy to remove it using other tools that Inspire Studio offers.

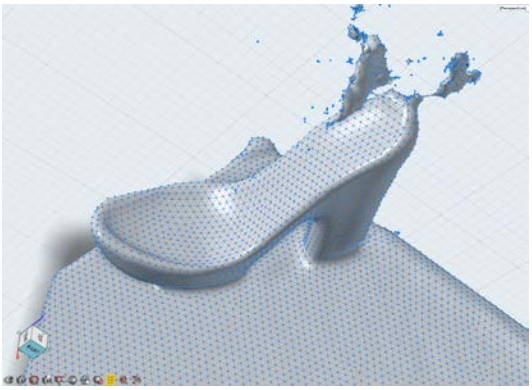
(a) Vertices of the reconstructed model.



(b) Zoom on a region of the previous image.

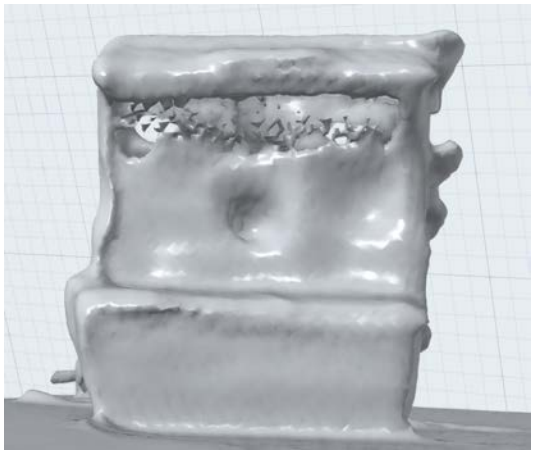


(c) The model after the simplification.



(d) Vertices of the simplified model.

Figure 12.2: We tried to reconstruct the model with a resolution of 500, and a voxel size of 0.6mm in each dimension, which resulted in more than 1 million vertices (in blue) and more than 2 million triangles. The third figure is a simplified model of the previous one, with 20 000 triangles, i.e. one hundredth of the original model. The simplified model is still understandable, it is just smoother and without the texture that it is possible to see on Figure 12.2.



(a) The front of the camera.

(b) One of the RGB frames.

Figure 12.3: This picture shows the front perspective of the reconstruction, and that holes for the lens and for the viewfinder have been detected. It is also possible to see that there is noise under the protective cover. A cause of this might be that some data could be missing, because we had to capture all the frames from the a certain angle, as the right image shows, to avoid including a glass surface on the background, which would have added a high quantity of noise.

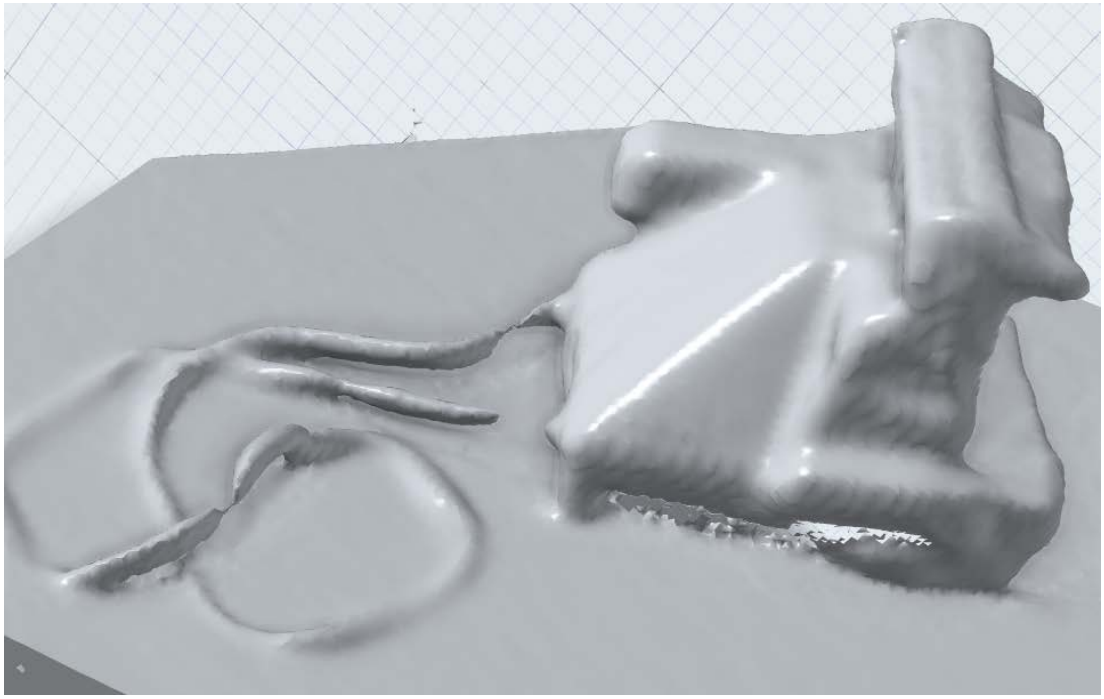
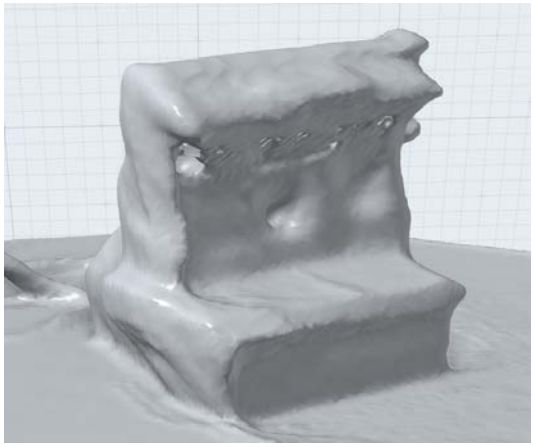
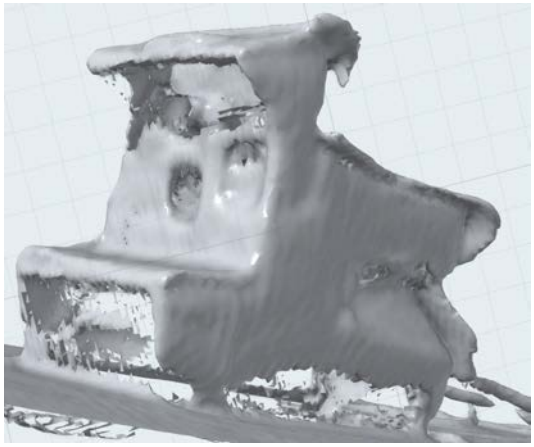


Figure 12.4: The side and back of the camera, and the attached string, which is very thin. The reconstruction contains also the shutter, but the area under has many holes.



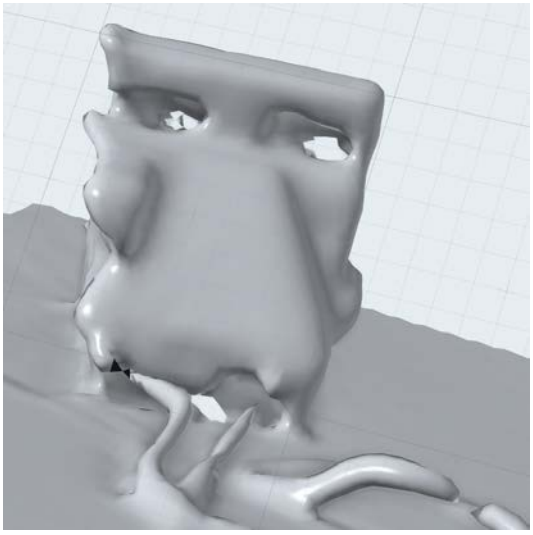
(a) Truncation value of 1.5cm.



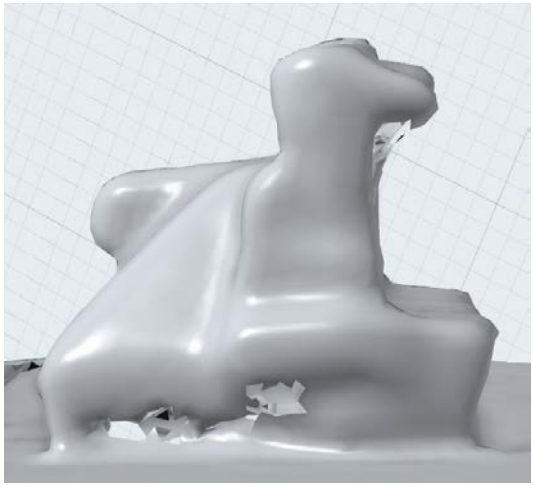
(b) Truncation value of 0.3cm.

Figure 12.5: This pictures show what changes by changing the value of signed distance function truncation. On the left, we can see that the holes of Figure 12.4, which had a truncation value of 1cm, have been closed, as well many the holes of the cover. On the contrary, on the right there are many holes, but edges are sharper.

(a) Front view.



(b) Back view, with also the string.



(c) Left view.

(d) Right view.

Figure 12.6: Simplification of the model to 20 000 triangles, which is about one tenth of the original triangle count. The simplified model is smoother than the original, that had a sort of textures, visible in all the previous pictures. Some of the details remain, it is the case of the lens and the string, as well as the problem with the edges of on the right view, whereas the hole for the viewfinder is evident. The simplification also filled some of the holes that were under the shutter.

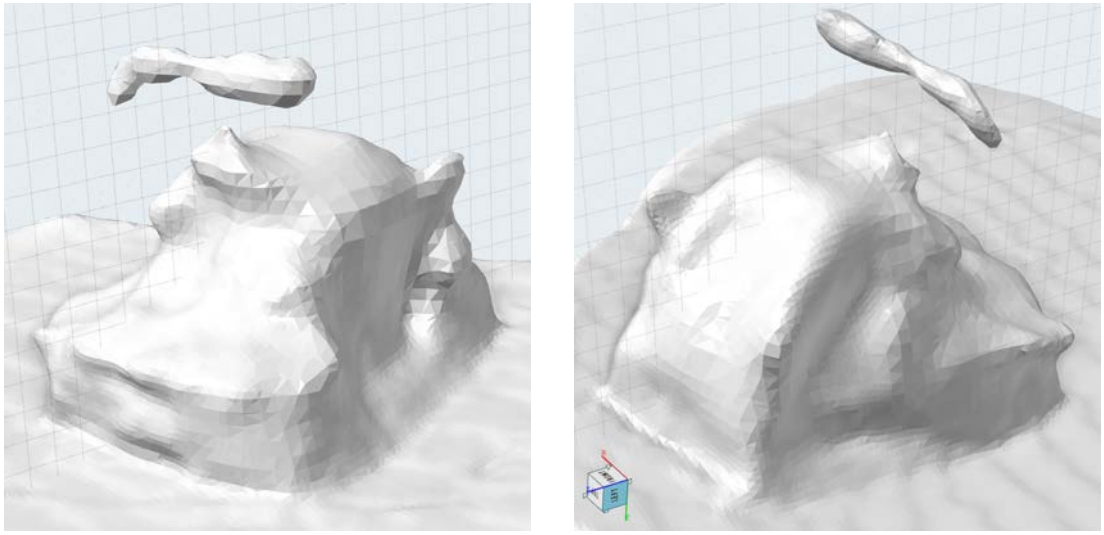
12.4 Comparison with Microsoft 3D Scan

3D Scan [22] is a free application, developed by Microsoft and available in the Windows 10 Store, that allows to scan objects and save a 3D model using a Kinect 2 or Kinect for Xbox One.

Its procedure is similar to ours: first, the user setups the data of volume to scan, then they scan the object by going around it, then the software creates of a point cloud from which it extract a triangle mesh.

However, 3D Scan does not limit the volume size: it asks for the maximum threshold of depth, which is used to filter the data that is acquired, but not to limit the volume, that is dynamic, instead. Moreover the software has a minimum threshold of depth fixed to 50cm, therefore it is likely for the volume to become big, which, accordingly to our experiments, might be a problem.

These differences are reflected in the setup step: it is performed on a 2D view, on which the user can set the width and height of the scan area, and the depth threshold is shown as color, i.e. what is within is colored, whereas what is outside is grayed. Setting the volume in the 3D scene is easier and clearer, in our opinion, although we can do it because we have a fixed volume.



(a) Front and right.

(b) Back and left.

Figure 12.7: The reconstruction of the instant camera from 3D Scan. Accordingly to our experience, the reason for this result was that the scan volume was too big.

Part IV: Conclusions and future works

Chapter 13

Conclusions

For this thesis we worked with and on the *plugin development kit* of Altair Inspire Studio. We managed to create a Python API that resembles the C++ existing one. We also designed an automated pipeline to keep it updated, requiring human intervention only to verify that it worked as expected, or to handle manually the few features based on incompatible differences between the languages.

We used the new API to create some example plugins, and we found that doing it starting from the C++ examples was quite immediate. We did not encounter problems, however the PDK exposes many functionalities, and we could not test all of them.

For the future, we would like to receive feedback from third parties, to understand what they think that could be improved.

We should also enhance the documentation, and we would like to create more involved example projects. A candidate is represented by the 3D acquisition plugin which we discussed about in the thesis.

This plugin is different from what Inspire Studio offers. It was helpful both as a starting approach to mew functionalities and to understand some of the technical limits of the PDK.

The resulting scans were not perfect as LiDAR ones can be, and they included some noise, but they were good enough for our purposes and for what RGBD sensors can do, considering also that they are much cheaper than LiDAR scanners.

In the future, we would like to resolve the problems we encountered, to improve the quality of the results, e.g. by computing the normals and using them as weight for the colors average, and to extend the supported devices and add advanced tuning of sensor-specific parameters.

Appendix A

Samples of code written with Python C API

A.1 Module creation

```c
static PyMethodDef mymodule_methods[] = {
  // Methods definitions will be here; no methods, for the basic example
  {NULL} // Sentinel, closes the list
};

static struct PyModuleDef mymodule = {
  PyModuleDef_HEAD_INIT, // A macro that initialize some fields to defaults
  "mymodule", // The name of module
  "Module documentation string", // May be NULL
  -1, // The module keeps state in global variables
  mymodule_methods // The array that contains the methods of the module
};

PyMODINIT_FUNC PyInit_mymodule(void)
{
  // Checks to the structures will be here, before the module initialization

  PyObject *m = PyModule_Create(&mymodule);
  if (!m) {
    // As described before, if a function fails, we should also fail
    return NULL;
  }

  // The rest of the registrations will be here

  return m;
}
```

Listing 10: The code to create a minimal module.

A.2 Custom type creation

```cpp
// The class we want to bind
class Example {
public:
  std::string cpp_string; // A C++ std::string we will bind with custom getter/setter
};

struct ExampleObject {
  PyObject_HEAD // Fields that allow the cast to a PyObject
  Example *real_object; // The real C++ instance of the class
  double double_val; // An example double value
  int int_constant; // An example integer constant
  const char *str_val; // An example to show that Python can manage strings
  PyObject *object; // An example of Python object without type enforcing
};

/* Create a new object.
Needed, otherwise the creation will be disabled (e.g. to use factories), however Python provides also a
↪  PyType_GenericNew function, which just calls tp_alloc.
This is called once, to create the object. */
static PyObject *Example_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
  ExampleObject *self;
  // type may be a subclass of Example
  self = (ExampleObject *) type->tp_alloc(type, 0);
  if (self != NULL) {
    // Constants usually are initialized on the new, accordingly to Python docs
    self->int_constant = 42;
    // Defer the rest of the initialization to tp_init
  }
  return self;
}

/* Initialize an object.
This can be called many times, exactly each time __init__ is called on the Python side. */
static int Example_init(ExampleObject *self, PyObject *args, PyObject *kwds)
{
  // For this example, ignore parameters, however they work as normal function, which we will see on
  ↪  next section

  /* The most critical part is the pointer to the C++ instance.
  We have to remember to delete it, but only if the new allocation does not fail. */
  Example *new_instance = new Example();
  if (!new_instance) {
    PyErr_NoMemory();
    return -1;
  }
  if (self->real_object) {
    delete self->real_object;
  }
  self->real_object = new_instance;

  // Initialize also all the other members

  // One should be careful also with PyObject *, because of the reference count
  PyObject *new_object = PyUnicode_FromString("hello");
```

```
54    PyObject *old_object = self->object;
55    self->object = new_object;
56    Py_XDECREF(old_object);
57
58    return 0;
59  }
60
61  // We have the pointer to the object, we need to delete it at finalization
62  static void Example_dealloc(ExampleObject *self)
63  {
64    delete self->real_object;
65    Py_TYPE(self)->tp_free((PyObject *) self);
66  }
67
68  // Getter for Example::cpp_string
69  static PyObject *Example_getcppstring(ExampleObject *self, void *closure)
70  {
71    return PyUnicode_FromString(self->real_object->cpp_string.c_str());
72  }
73
74  // Setter for Example::cpp_string
75  static int Custom_setcppstring(ExampleObject *self, PyObject *value, void *closure)
76  {
77    if (value == NULL) {
78      PyErr_SetString(PyExc_TypeError, "Cannot delete the cpp_string attribute");
79      return -1;
80    }
81    if (!PyUnicode_Check(value)) {
82      PyErr_SetString(PyExc_TypeError, "The cpp_string attribute value must be a string");
83      return -1;
84    }
85    const char *string = PyUnicode_AsUTF8(value);
86    if (!string) {
87      return -1;
88    }
89    self->real_object->cpp_string = string;
90    return 0;
91  }
92
93  // The members of ExampleObject that Python can manage directly
94  static PyMemberDef Example_members[] = {
95    // name, type, offset in the struct, flag (can be 0 or READONLY), documentation string
96    {"double_val", T_DOUBLE, offsetof(ExampleObject, double_val), 0, "A double value"},
97    {"int_constant", T_INT, offsetof(ExampleObject, int_constant), READONLY, "An integer constant"},
98    {"str_val", T_STRING, offsetof(ExampleObject, str_val), 0, "A (readonly) C string"}, // T_STRING
      ↪  implies READONLY
99    {"object", T_OBJEXT_EX, offsetof(ExampleObject, object), 0, "Any Python object"},
100   {NULL} // Sentinel
101 };
102
103 // Casts to getter and setter are needed because they take their specialized object, whereas
    ↪  PyGetSetDef expects functions taking generic PyObject *
104 static PyGetSetDef Example_getsetters[] = {
105   // name, getter, setter, documentation, value to pass as closure parameter to getters and setters
106   {"cpp_string", (getter) Example_getcppstring, (setter) Example_setcppstring, "Documentation for
      ↪  cpp_string", NULL},
107   {NULL}  // Sentinel
108 };
109
```

```
110 static PyMethodDef Example_methods[] = {
111   {NULL} // Sentinel, no methods for the example
112 };
113
114 static PyTypeObject ExampleType = {
115   PyVarObject_HEAD_INIT(NULL, 0)
116   .tp_name = "mymodule.Example", // The qualified name of the type
117   .tp_doc = "Binding of the Example class plus other example members", // Documentation string
118   .tp_basicsize = sizeof(ExampleObject), // Size of the object, for the default allocator function
119   .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, // Allow this class to have children
120   .tp_new = Example_new,
121   .tp_init = (initproc) Example_init,
122   .tp_dealloc = (destructor) Example_dealloc,
123   .tp_members = Example_members,
124   .tp_methods = Example_methods,
125   .tp_getset = Example_getsetters,
126 };
127
128 PyMODINIT_FUNC PyInit_mymodule(void)
129 {
130   // Checks for the validity of the structure and set some defaults
131   if (PyType_Ready(&ExampleType) < 0)
132     return NULL;
133
134   PyObject *m = PyModule_Create(&mymodule);
135   if (!m) {
136     return NULL;
137   }
138
139   // Finally register the example type
140   Py_INCREF(&ExampleType);
141   PyModule_AddObject(m, "Example", (PyObject *) &ExampleType);
142
143   return m;
144 }
```

Listing 11: The code to create a custom type. For clarity of exposure, in this example we use *direct initialization*; although it is standard in C, it will be standardized only in the next version of C++ (known as C++20, at the moment).

A.3 Exposing a function

```c
// The convention is to call the function modulename_functionname.
// All these helper functions do not need to be called from outside, therefore they can be static
static PyObject *module_custom_method(PyObject *self, PyObject *args, PyObject *kw)
{
  // Declare variables to hold arguments
  PyObject *arg1;
  char *arg2;
  int arg3;
  ExampleObject *arg4;
  // etc

  // Define the keywords and unpack the arguments
  char *keywords[] = {"kw1", "kw2", /* etc */};
  if (!PyArg_ParseTupleAndKeywords(args, kw, "format", keywords, &arg1, &arg2, &arg3, &arg4 /*, etc */)) {
    // As always, return NULL if it fails; the exception is already set by PyArg_ParseTupleAndKeywords
    return NULL;
  }

  // Process the arguments, if needed, e.g. check for validity of the domain, do some cast
  if (arg3 < 0) {
    PyErr_SetString(PyExc_ValueError, "The third argument cannot be negative");
    return NULL;
  }

  // Call the actual function
  real_function(arg2, arg3, arg4->real_instance);

  // Cast the return value and return it
  // return PyLong_FromLong(123); // E.g. this returns a Python int
  // return PyFloat_FromDouble(123.4); // This returns a Python float
  /* or */ Py_RETURN_NONE; // If a C function returns void, the callback has return None, as NULL is used only for exceptions
}

static PyMethodDef methods[] = {
  {
    "custom_method", // Method name
    module_custom_method, // Pointer to the callback
    METH_VARARGS | METH_KEYWORDS, // Our callback accepts both varargs and kwargs
    "Documentation string for the function"
  },
  {NULL} // Sentinel
};
```

Listing 12: The code to expose a function.

Appendix B

Example of a Clang AST

```
class A {
public:
  class B {
    void method1() {}
  };
  typedef std::vector<B> bvec;
  enum C {v1, v2, v3};
  int method2(int a, int b)
  {
    return a + b;
  }
  int mMember;
};
```

```
class A definition
  |-DefinitionData pass_in_registers aggregate standard_layout trivially_copyable pod trivial literal
  | |-DefaultConstructor exists trivial needs_implicit
  | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveConstructor exists simple trivial needs_implicit
  | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveAssignment exists simple trivial needs_implicit
  | `-Destructor simple irrelevant trivial needs_implicit
  |-CXXRecordDecl 0x31ede38 <col:1, col:7> col:7 implicit class A
  |-AccessSpecDecl 0x31edec0 <line:3:1, col:7> col:1 public
  |-CXXRecordDecl 0x31edee8 <line:4:2, line:6:2> line:4:8 referenced class B definition
  | |-DefinitionData pass_in_registers empty aggregate standard_layout trivially_copyable pod trivial
  ↪  literal has_constexpr_non_copy_move_ctor can_const_default_init
  | | |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
  | | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | | |-MoveConstructor exists simple trivial needs_implicit
  | | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
  | | |-MoveAssignment exists simple trivial needs_implicit
  | | `-Destructor simple irrelevant trivial needs_implicit
  | |-CXXRecordDecl 0x31edff8 <col:2, col:8> col:8 implicit class B
  | `-CXXMethodDecl 0x31ee0c8 <line:5:3, col:19> col:8 method1 'void ()'
  |   `-CompoundStmt 0x31eea28 <col:18, col:19>
  |-TypedefDecl 0x31ee578 <line:7:2, col:25> col:25 bvec 'std::vector<B>':'std::vector<A::B,
  ↪  std::allocator<A::B> >'
  | `-ElaboratedType 0x31ee4c0 'std::vector<B>' sugar
  |   `-TemplateSpecializationType 0x31ee480 'vector<A::B>' sugar vector
  |     |-TemplateArgument type 'A::B'
  |     `-RecordType 0x31ee460 'std::vector<A::B, std::allocator<A::B> >'
  |       `-ClassTemplateSpecialization 0x31ee370 'vector'
  |-EnumDecl 0x31ee5c8 <line:8:2, col:20> col:7 C
  | |-EnumConstantDecl 0x31ee680 <col:10> col:10 v1 'A::C'
  | |-EnumConstantDecl 0x31ee6d0 <col:14> col:14 v2 'A::C'
  | `-EnumConstantDecl 0x31ee720 <col:18> col:18 v3 'A::C'
  |-CXXMethodDecl 0x31ee900 <line:9:2, line:12:2> line:9:6 method2 'int (int, int)'
  | |-ParmVarDecl 0x31ee788 <col:14, col:18> col:18 used a 'int'
  | |-ParmVarDecl 0x31ee800 <col:21, col:25> col:25 used b 'int'
```

```
35    | `-CompoundStmt 0x31eead8 <line:10:2, line:12:2>
36    |   `-ReturnStmt 0x31eeac8 <line:11:3, col:14>
37    |     `-BinaryOperator 0x31eeaa8 <col:10, col:14> 'int' '+'
38    |       |-ImplicitCastExpr 0x31eea78 <col:10> 'int' <LValueToRValue>
39    |       | `-DeclRefExpr 0x31eea38 <col:10> 'int' lvalue ParmVar 0x31ee788 'a' 'int'
40    |       `-ImplicitCastExpr 0x31eea90 <col:14> 'int' <LValueToRValue>
41    |         `-DeclRefExpr 0x31eea58 <col:14> 'int' lvalue ParmVar 0x31ee800 'b' 'int'
42    `-FieldDecl 0x31ee9c0 <line:13:2, col:6> col:6 mMember 'int'
```

Listing 13: An example of some C++ declaration and the corresponding AST, dumped by running Clang with the option `-ast-dump`.

Bibliography

[1] David Abrahams and Stefan Seefeld. *Boost.Python*. URL: https://www.boost.org/doc/libs/1_71_0/libs/python/doc/html/index.html (visited on 2019-08-26).

[2] David M. Beazley. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++". In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*. TCLTK'96. Monterey, California: USENIX Association, 1996, pp. 15–15.

[3] Matthew Berger et al. "A Survey of Surface Reconstruction from Point Clouds". In: *Computer Graphics Forum* (2016), p. 27. DOI: 10.1111/cgf.12802. URL: https://hal.inria.fr/hal-01348404.

[4] Paul J. Besl and Neil D. McKay. "A method for registration of 3-D shapes". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992-02), pp. 239–256. DOI: 10.1109/34.121791.

[5] Paul Bourke. *Polygonising a scalar field*. URL: http://paulbourke.net/geometry/polygonise/ (visited on 2019-09-17).

[6] Shirit Brook et al. "D400 Series Visual Presets". In: *librealsense Wiki*. URL: https://github.com/IntelRealSense/librealsense/wiki/D400-Series-Visual-Presets (visited on 2019-09-26).

[7] The Clang Team. *LibTooling*. URL: https://clang.llvm.org/docs/LibTooling.html (visited on 2019-08-27).

[8] Brian Curless and Marc Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237269.

[9] Cameron Desrochers. *A single-producer, single-consumer lock-free queue for C++*. URL: https://github.com/cameron314/readerwriterqueue (visited on 2019-09-15).

[10] Isaac Gouy. *The Computer Language Benchmarks Game*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gpp.html (visited on 2019-08-22).

[11] Anders Grunnet-Jepsen. *Tuning D435 and D415 cameras for optimal performance*. 2018.

[12] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521540518.

[13] James Hays. *Stereo intro and Camera Calibration*. URL: https://www.cc.gatech.edu/~hays/compvision/lectures/09.pdf (visited on 2019-09-11).

[14] Intel. *Intel® RealSense™ Camera Depth Testing Methodology*. URL: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_DepthQualityTesting.pdf (visited on 2019-09-12).

[15] Shahram Izadi et al. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: ACM, 2011, pp. 559–568. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047270.

[16] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 — Seamless operability between C++11 and Python*. URL: https://github.com/pybind/pybind11 (visited on 2019-08-26).

[17] Wenzel Jakob et al. "Instant Field-Aligned Meshes". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)* 34.6 (2015-11). DOI: 10.1145/2816795.2818078.

[18] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422.

[19] Sergey Lyskov. *Binder*. URL: https://github.com/RosettaCommons/binder (visited on 2019-08-27).

[20] Sergey Lyskov. "PyRosetta-4". In: *RosettaCon*. 2016. URL: https://graylab.jhu.edu/RosettaCon2016/PyRosetta-4.pdf (visited on 2019-08-26).

[21] Microsoft Corporation. "/bigobj (Increase Number of Sections in .Obj file)". In: *Visual Studio 2015 Documentation*. URL: https://docs.microsoft.com/en-us/cpp/build/reference/bigobj-increase-number-of-sections-in-dot-obj-file?view=vs-2015 (visited on 2019-08-30).

[22] Microsoft Corporation. *3D Scan*. URL: https://www.microsoft.com/en-us/p/3d-scan/9nblggh68pmc (visited on 2019-09-20).

[23] Richard A. Newcombe et al. “KinectFusion: Real-time dense surface mapping and tracking”. In: *2011 10th IEEE International Symposium on Mixed and Augmented Reality*. 2011-10, pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378.

[24] The OpenCV Team. “cv::kinfu::KinFu Class Reference”. In: *OpenCV 4.1.1 Documentation*. URL: https://docs.opencv.org/4.1.1/d8/d1f/classcv_1_1kinfu_1_1KinFu.html (visited on 2019-09-15).

[25] Joshua Partlow. “Write C++ extensions for Python”. In: *Visual Studio 2017 Documentation*. URL: https://docs.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio (visited on 2019-08-27).

[26] Python Software Foundation. “History and License”. In: *Python 3 Documentation*. URL: https://docs.python.org/3/license.html (visited on 2019-09-20).

[27] Python Software Foundation. “Modules”. In: *Python 3 Documentation*. URL: https://docs.python.org/3/tutorial/modules.html (visited on 2019-08-23).

[28] Python Software Foundation. *Python/C API Reference Manual*. URL: https://docs.python.org/3/c-api/index.html (visited on 2019-08-22).

[29] Python Software Foundation. *Quotes about Python*. URL: https://www.python.org/about/quotes/ (visited on 2019-08-22).

[30] Szymon Rusinkiewicz and Marc Levoy. “Efficient variants of the ICP algorithm”. In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. 2001-05, pp. 145–152. DOI: 10.1109/IM.2001.924423.

[31] The SciPy Community. “Linear algebra”. In: *NumPy v1.17 Manual*. URL: https://docs.scipy.org/doc/numpy/reference/routines.linalg.html (visited on 2019-08-22).

[32] Maksim Shabunin, Ryan Fox, and Alexander Alekhin. “Depth Map from Stereo Images”. In: *OpenCV 4.1.1 Documentation*. URL: https://docs.opencv.org/4.1.1/dd/d53/tutorial_py_depthmap.html (visited on 2019-09-11).