TU WIEN Informatics

High-Performance Framework for Dataset Generation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Maximilian Riegler
Matrikelnummer 01634877

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Philipp Erler

Wien, 5. August 2020

Maximilian Riegler Michael Wimmer

Informatics

High-Performance Framework for Dataset Generation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Maximilian Riegler
Registration Number 01634877

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Philipp Erler

Vienna, 5th August, 2020

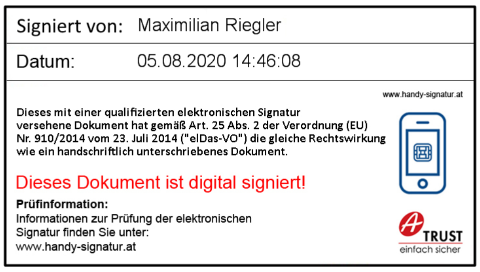
Maximilian Riegler	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Maximilian Riegler

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. August 2020



Signiert von:	Maximilian Riegler
Datum:	05.08.2020 14:46:08

www.handy-signatur.at

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr. 910/2014 vom 23. Juli 2014 ("elDas-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

TRUST
einfach sicher

Kurzfassung

Das Ziel dieser Bachelorarbeit ist die Entwicklung eines Python Frameworks. Die Hauptaufgabe des Frameworks ist die Generierung von Datensätzen, welche für Oberflächenrekonstruktion genutzt werden. Diese werden für das Trainieren eines Neuronalen Netzwerkes benötigt, welches in der Lage ist, ein 3D-Modell anhand einer gegebenen Punktwolke wiederherstellen kann. Um das Training des Neuronalen Netzes zu optimieren, werden eine Menge an Trainingsdaten benötigt. Dieses Framework nutzt Multi-Processing, um einen schnelleren Generierungsprozess, im Vergleich zur sequentiellen Generierungen mehrerer 3D-Modelle nacheinander, zu erreichen.

Zusätzlich ist das Framework in der Lage beliebige ähnliche Pipelines zu handhaben. Die BenutzerIn ist in der Lage die einzelnen Schritte so einer Pipeline in einem XML Dokument zu definieren, welche in der Lage sind, beliebige Programme aufzurufen. Dieser Punkt macht dieses Framework zu einem Allzweckwerkzeug für jegliche Aufgaben, bei denen eine Menge an Daten unabhängig voneinander bearbeitet werden müssen.

Die Ergebnisse zeigen einen großen Performance-Gewinn beim Generieren von Datensätzen. Dies spiegelt sich in den durchgeführten Benchmarks wieder. Dabei wurde die Ausführungszeit für eine fixe Menge an Dateien in verschiedenen Ausführungsmodi gemessen. Der eigens entwickelte Prozesspool zeigt schnellere Zeiten als die Nutzung von Python's Prozesspool, welcher jeden Schritt der Pipeline unabhängig voneinander bearbeitet. Er ist zudem wesentlich schneller, als jeden Schritt für jede Datei sequentiell auszuführen.

Abstract

The aim of this bachelor thesis is the development of a Python framework. The main task for this framework is the generation of datasets, which can be further used for surface reconstruction. They are needed for training a neural network, which is then able to reconstruct meshes on its own given a point cloud of a mesh. In order to optimize the training of the neural network, a lot of training data is needed. This framework utilizes multi-processing to achieve a faster generation process in comparison to sequentially generating one mesh after another.

In addition, the framework is also able to handle any kind of similar pipeline. The user is able to define the steps of such pipeline in an XML document, which then can make calls to arbitrary programs. This fact makes the framework an all-purpose tool for any kind of task that needs to process a lot of data independent from each other.

The results show a great performance increase when generating datasets. This can be seen in the benchmarks that have been done. The time of execution for a fixed amount of files has been measured with different modes of execution. The custom process pool we developed shows a faster time overall compared to using Python's process pool for each step of the pipeline independently. It is also way faster in comparison to running every step for each file sequentially.

Contents

CHAPTER 1

Introduction

Pipelines can be found in various areas of computer science. In general, a pipeline consists of multiple components, where the output of one is the input of the next in line. In the context of a software pipeline, this means that one program gives its output to another one and so on until we successfully calculate the result.

Looking at the field of AI and neural networks, supervised learning requires training data that consists of input-output pairs. Preparing such tuples often needs multiple steps, which can be viewed as such a software pipeline. The focus of this bachelor thesis is to help with the generation of such training data in the field of surface reconstruction. Given a point cloud, such a network approximates a mesh's surface. In order to train a neural network, which can achieve this task, we need complete meshes and point clouds that we create from them.

1.1 Problem Statement

In order to train neural networks in the area of surface reconstruction, datasets consisting of point clouds need to be generated using available meshes. The process of producing such datasets consists of multiple steps, which can be tedious and error-prone if done manually. In addition, generating datasets for many meshes can be a very time-consuming task. We can accelerate this with multi-processing but need to be careful with overhead. Also, this pipeline must to be easily adaptable, with little to no effort from the user and without changing the framework code.

1.2 Aim of this Project

The aim of this bachelor's thesis is the development of a Python framework that can generate datasets for use in surface reconstruction. The existing code already uses multi-processing paradigms. However, it proves to be a bit confusing because of duplicate code.

The focus now lies in keeping high performance, but to generalize the multi-processing to avoid such code duplication.

Furthermore, the framework needs to be abstract in a way that it can be used outside the dataset generation context. The users can define their own pipeline in a simple XML document consisting of edges of a graph. As a result, we construct the pipeline as a graph where each edge represents a call to a given program. On the other hand, the nodes represent folders where we save the pipeline input, intermediate files and the final result. This graph-based approach has the advantage that it can be easily visualized. We can use graph-based algorithms to e.g. check dependency cycles and allow a depth-first processing.

1.3 Dataset Generation Pipeline

The specific pipeline for the generation of datasets used in this thesis consists of 6 steps. In order to provide a quick overview of the complexity of these steps, we summarize them in the following section. More about the theory and what these datasets are actually used for, can be found in Points2Surf by Erler et al. [EGO+20].

1.3.1 Convert Meshes

The first step in the pipeline is to convert every mesh of the dataset into a common file format. There are around five thousand in this dataset that we take from the ABC-Dataset project [KMJ+19] (see Chapter 2). We convert them to the .ply (Stanford triangle) format in order to provide a clean and explicit representation of the meshes. The concept of such explicit surfaces is rather simple: Each mesh consists of a list of its vertices containing its coordinates. Each such vertex can also save additional information like UV-coordinates and normals. In case of a .ply file, after the list of vertices comes a list of faces of the mesh.

1.3.2 Clean Meshes

In order to calculate signed distances, which we need in a later step of the pipeline, the meshes need to be very clean. Besides some smaller corrections, this step serves to filter meshes that do not fulfill certain properties. The meshes need to represent solids:

- closed and watertight: They do not have any holes in the surface.
- two-manifold: Each edge must be incident to exactly two faces.
- without self-intersections: There should not be any self-intersections within the mesh.

1.3.3 Scale Meshes

This step scales the meshes to the same range. In machine learning context, we call this normalization. We use it, especially when the data available has different ranges, to make the learning process of the neural network more efficient. Therefore, we scale the meshes in such a way that the longest side of the bounding box is between -1 and 1. An additional advantage of that range is the fact that floating point numbers provide a better precision around zero.

1.3.4 Sampling with Blensor

The aim of this step is to create point clouds from the given meshes in a realistic way. For this purpose, we use the BlenSor [GKUP11] project, which is capable of simulating various kinds of range scanners. The sensor of special interest in this step is the time-of-flight camera that is finally used to generate the point clouds. An important factor for the generation are different kinds of artifacts and errors that add a further layer of realism. Non-uniform sampling and noises are just two types of measuring artifacts that occur in real world scenarios. Therefore, the neural network must be able to handle them as well, which is why we enforce them to a certain degree in the point cloud generation.

1.3.5 Get Query Points

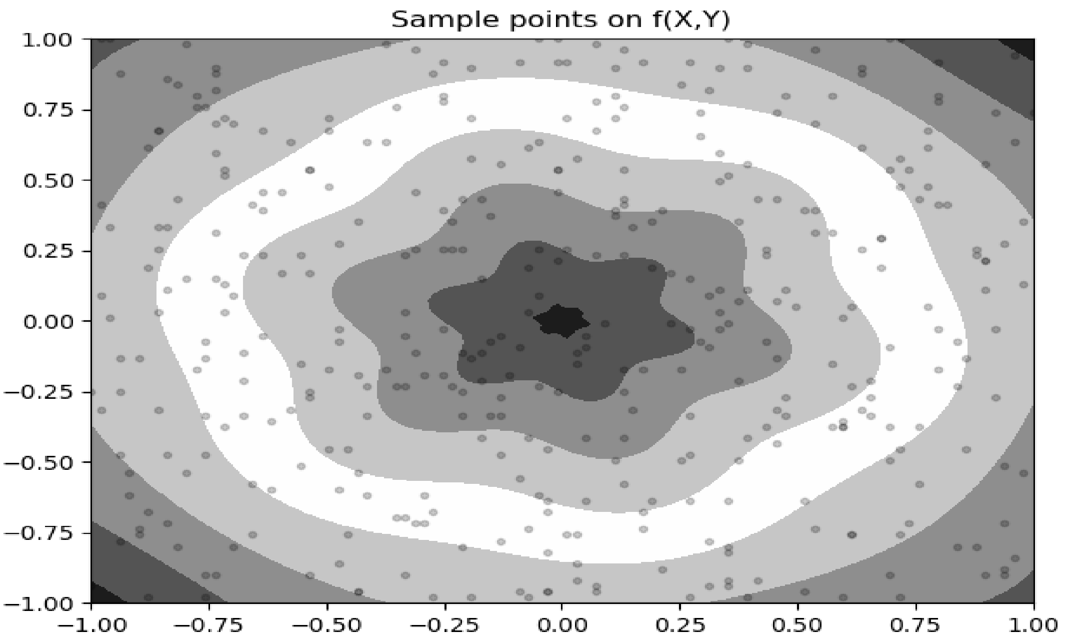


Figure 1.1: Example how query points can be used in 2D. Graphic shows selection of query points. The white sections can be seen as the surface of a mesh.

This step serves two purposes. At first, we convert the explicit geometry from the meshes to an implicit form. It gets converted to a Signed Distance Field (SDF). An SDF in short, is a volume in which every point defines the signed distance to the nearest surface of the actual mesh. Now the calculation of the real signed distance only gets done for a selection of points (query points) and serves as the ground truth for the neural network. It uses these query points for learning and is then able to calculate the whole signed distance field from a given point cloud. After the inference, it reconstructs the explicit mesh using the Marching Cubes algorithm [LLVT03].

1.3.6 Making Dataset Splits

This final step splits the dataset into two subsets of training and test data. We use the training set to train the neural network, whereas the test set is for evaluating the results the network produces. The result of this step are two corresponding text files containing the names of the point clouds that need to be loaded into the neural network.

CHAPTER 2

Related Work

This chapter will take a quick peek at a couple of different papers related to similar approaches and ideas like in this thesis.

In the context of deep learning in the domain of surface reconstruction, it is necessary to have a rich set of different shapes and meshes in order to train a network. As one of the main purposes for the framework developed during this thesis is such task, it makes sense to look at available work. The ABC-Dataset project [KMJ+19] is a large repository of Computer-Aided Design (CAD) models containing information like e.g. explicitly parameterized curves. This data can be used as a ground truth when it comes to training neural networks or specific shape reconstruction from point clouds. The collection includes one million models in total, which they acquired over a time period of four months from another CAD collection named Onshape. However, these models do not contain the needed properties, which is why ABC-Dataset put its own pipeline for further refining this data into place. This pipeline relies heavily on free software to provide more engagement through the community and is also built to be run in parallel on big clusters. Unfortunately, it is not specified how exactly it can be parallelized efficiently.

Another example for a collection of meshes is ShapeNet [CFG+15]. The goal of this project is pretty similar to other comparable repositories. It gathers raw 3D models from various publicly available resources and then annotates them in different steps. These semantic annotations range from being language-, geometric- or even physical-related. The approach for the acquisition of such rich annotations is not exclusively done by hand. Some types of annotations can be predicted algorithmically and are later checked by human experts or crowd-sourcing pipelines.

Besides the purpose of this framework, generating datasets, the structure of it is another special aspect. The idea is to build a graph with edges and nodes that respectively represent the program calls and folders. In *A graph-based approach to Web services composition* by Hashemian et al. [HM05], they use a similar graph-based method in another domain. There, they compose complex web services using a bunch of smaller web services. They put the information about the known services into a dependency graph, which then gets processed with algorithms for finding the perfect composition. Within these algorithms they use other well-known techniques like breadth-first search. Similarly, we apply depth-first search in this bachelor thesis to achieve quick complete results while parallelizing the whole pipeline. More about this topic can be found in the implementation of the custom process pool in the Chapter 3.3.6.

CHAPTER 3

Method

This chapter is about the actual implementation of the framework. The first two sections contain definitions, general information and brief descriptions of the external libraries we use. Furthermore, the third section provides descriptions and explanations of the different components of the framework. High performance, especially when processing large amounts of files, is one key aspect of this framework.

3.1 Definitions and General Information

An edge in this context is a part of the pipeline, which processes input files and produces output files. We use the term edge here, because we store the pipeline internally as a graph.

A node represents an actual folder in the file system. We connect them via an edge. Nodes at the beginning of edges are input directories and nodes at the end are output directories.

In contrast to the graph elements like edges and nodes, there are the actual processing steps. The graph provides the general structure of the pipeline. The code follows the correct order of edges and does some kind of work for each one. That includes checking the available input files and if there are maybe already the correct output files. We do this via comparing the time stamps of input and output files. The final and most important step, which we do for each edge, is to make the program call. A call means to start the binary specified for the current edge, which does the actual work of transforming an input to an output file. If the framework now runs with multiple input files, it will make a call to the specified program every time such a file reaches that edge. This means one edge can make many calls during execution, depending on the number of files to be processed.

Another important aspect of this framework is multi-processing. We use multi-processing for the implementation of the custom process pool and in order to spawn new sub-processes for program calls. An alternative would be to use multi-threading. However, there are some limitations when it comes to multi-threading in Python. The so-called Global Interpreter Lock is a mechanism for making the Python Interpreter thread-safe. It is a mutex that prevents multiple threads to simultaneously access the interpreter. It was first introduced to Python in order to avoid race conditions e.g. when manipulating the reference count of objects. The disadvantage of the Global Interpreter Lock is especially noticeable with CPU-heavy programs. Multiple threads would essentially become useless, because only one can be effectively used at a time. Therefore, to avoid problems with the Global Interpreter Lock, we use multi-processing in this framework instead [Pytb].

3.2 External Libraries

There are two types of libraries that we use in the thesis. One kind we use for the different steps of the dataset generation pipeline. These have no direct relation to the framework, as it depends on which actual pipeline the user defined and therefore, we do not explain them any further. The other ones are necessary for the framework to function correctly.

Framework libraries:

- networkx[1] (2.4): Provides a data structure for graphs in which we store the processing pipeline. In addition, it provides methods for traversal (DFS, BFS), checking for properties (cyclic?) and visualization.
- pygraphviz[2] (1.5): Dependency of networkx for drawing graphs.
- tqdm[3] (4.46.1): Progress bar to visualize how many calls in the pipeline are already done.
- xmlschema[4] (1.2.0): Support for XML Schema. It checks if the pipeline configuration is valid.

3.3 The Framework

This section provides an overview how working with the framework might look like and which components we implemented.

[1]https://networkx.github.io/
[2]https://pygraphviz.github.io/
[3]https://github.com/tqdm/tqdm
[4]https://pypi.org/project/xmlschema/

3.3.1 Configuration

The first step in working with the framework is to have a correctly formatted configuration file. We made the configuration using XML and it needs to have a certain structure that we check using an XML schema file (see Chapter 5.1). The reason for using the XML format is that on one hand, it enables the user to easily write their own extensive configuration in a simple manner. On the other hand, there is support in many different programming languages to parse such XML files, like in this case Python. An example configuration may look like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<edge-set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="edge-definition.xsd" version="0.2">
  <edge type="regular">
    <name>Convert meshes</name>
    <description>Converts meshes in another format using trimesh
    </description>
    <inputDir>~/00_base_meshes</inputDir>
    <outputDir>~/01_base_meshes_ply</outputDir>
    <call>python ~/convert_mesh.py</call>
    <parameters>
      <option name="-t" value=".ply"/>
      <argument type="input"/>
      <argument type="output"/>
      <argument value="example"/>
    </parameters>
  </edge>
</edge-set>
```

The root element “edge-set” contains multiple (in this example just one) “edge” elements. This edge connects multiple input (“inputDir”) and output (“outputDir”) folders together. Another necessary part of the definition is the actual “call” that contains the program call without any command line arguments. These must be defined separately in the “parameters” block. There are options and arguments. Options have a flag name and an optional value while arguments can have two different kinds of types:

- value: A simple fixed value.
- input/output: The program, which gets called, somehow needs to know with which files it must work. We do this using arguments; however, these should not be hard-coded in this configuration. The input/output type serve as placeholders for the defined “inputDir” and “outputDir”. The framework replaces them then at execution time (see Chapter 3.3.2).

The "config_parser.py" module is responsible for working with such a configuration. Its purpose is to first validate the configuration against the XML schema [5]. We do this via the xmlschema library (for reference see Chapter 3.2). Next it parses the whole file with the Python ElementTree XML API and stores the edges as a graph. We do further validation, like checking for cycles in the graph, before returning it to the caller.

3.3.2 Building Calls and Execution

During the parsing of the configuration, each edge gets a basic call string with placeholders. We create such string using the following code:

```
for p in edge_xml.find("parameters"):
  if p.tag == "option":
    call += " " + p.attrib["name"]

  if "value" in p.attrib and "type" not in p.attrib:
    call += " " + p.attrib["value"]
  elif "type" in p.attrib and "value" not in p.attrib:
    call += " {}"
    format_order.append((IOType.INPUT, inputs.pop(0))
                        if p.attrib["type"] == "input"
                        else (IOType.OUTPUT, outputs.pop(0)))
```

The loop iterates over every child of "parameter" in the XML configuration. Flags and fixed values get directly put into the string. In case of a dynamic input or output we use a placeholder string "{}". The format_order list saves additional information about these parameters. It saves the exact sequence of the dynamic parameters, so we can later know which placeholder string needs to be filled with an input or output.

Once the framework runs and an actual call needs to be made in an edge, the following code builds the concrete call string:

```
def make_program_call(self, file, inputs, input_files,
                      outputs, inputs_available=True,
                      logging_queue=None):
  # Fetch the basic call command
  program_call = self.get_call()

  i = 0
  o = 0
  output_folders = []
  input_files_dict = {}
  params = []
  for (format_type, _) in self.get_format_order():
    if inputs_available and format_type == IOType.INPUT:
      input_file = ""
      input_directory = ""
      if i == 0:
        input_file = file
```

[5]XML schema specification `https://www.w3.org/TR/xmlschema11-1/`

```
        input_directory = inputs[0]
      else:
        input_file = ExecutionModelHelper.find_best_match(file,
                     input_files[i])
        input_directory = inputs[i]
      params.append(path.join(input_directory, input_file))
      if input_directory in input_files_dict.keys():
        input_files_dict[input_directory].append(input_file)
      else:
        input_files_dict[input_directory] = [input_file]
      i += 1
    elif format_type == IOType.OUTPUT:
      params.append(outputs[o])
      output_folders.append(outputs[o])
      o += 1

  for o in output_folders:
    if not path.exists(o):
      makedirs(o, exist_ok=True)

  # Insert the parameters in the placeholders of the call
  program_call = program_call.format(*params)
  # Execute command
  return ExecutionModelHelper.run_command(program_call, self.get_name(),
                    None if not input_files_dict else input_files_dict,
                    output_folders, logging_queue=logging_queue)
```

The for-loop iterates over the format order of the edge and inserts, depending on the type, the corresponding input or output directory to a list called params. When this finishes the original call with the placeholders gets formatted with the params list. After that, we execute it in the run_command method.

```
def run_command(command, edge, input_files,
output_folders, logging_queue=None):
  args = command.split(" ")

  output = None
  returncode = None

  if args[0] == "python":
    output = StringIO()
    try:
      sys.stdout = output
      name = path.splitext(path.basename(args[1]))[0]
      location = args[1]
      called_module_spec = spec_from_file_location(name, location)
      if called_module_spec is not None:
        called_module = module_from_spec(called_module_spec)
        called_module_spec.loader.exec_module(called_module)
        called_module.main(args[2:])
    except AttributeError as ex:
      print("Missing main method!")
      returncode = 1
    except Exception as ex:
      returncode = 1
    else:
      returncode = 0
    finally:
      sys.stout = sys.__stdout__
      output = output.getvalue()
  else:
    process = subprocess.run(args, stdout=subprocess.PIPE,
              stderr=subprocess.STDOUT, universal_newlines=True)
    output = process.stdout
    returncode = process.returncode

  created_files = ExecutionModelHelper.extract_created_files(
                output_folders)
  logging_information = (edge, input_files, created_files,
                       returncode, output)
  if logging_queue is None:
    LoggingManager.log_process_execution(*logging_information)
  else:
    logging_queue.put(logging_information)

  return created_files
```

We do the execution of calls in two different ways. Depending if we make the call to another Python script or an arbitrary program. We do the call to a Python script by dynamically loading it as a module and running its main method. This achieves a huge performance boost, especially under Windows systems (more details in Chapter 3.3.6). In case that we call any other program, the framework spawns a sub-process and runs it there.

3.3.3 Shell

There are multiple ways to use the framework. The whole package can be imported into a custom Python project to use the classes directly. However, the intended way is to call the main file. There, the config paths can be set and the whole pipeline runs exactly once. Alternatively, you can run the main file with the -i flag. This will open an interactive shell instead, from where the framework can be started. It also provides some additional functionality like:

- Being able to visualize the pipeline.
- Some functions to access certain information from the log.

3.3.4 Pipeline Visualization

Using the shell, the user is able to export the defined pipeline as a .png file. This provides a good overview of which dependencies between the different steps exist. An example can be seen at Figure 3.1. Each node represents a path/folder and each edge a program call.

3.3.5 Execution Models

The framework provides in total 3 different modes of operations or "execution models". These modes change the behavior of how exactly the pipeline gets processed.

The first model is a sequential order of execution. The process starts with the first edge and makes the program call for each input file one after another. Once it finishes the whole edge, it moves on to the next one. We implemented an own iterator for the correct sequence of edges. It takes dependencies into account, because some edges need to be processed before some other.

The next model uses the process pool provided by Python. Each edge is, again, processed in the same order as in the sequential model. However, inside each edge a pool gets utilized. The pool splits up the list of input files into even chunks. The size of these chunks depends on the number of virtual processors available. Each chunk is then assigned to one sub-process that works independent of the other processes [Pytc].

The last model that we realized is an own variant of a process pool. The idea behind it is to get a result from each edge as quickly as possible. The pool now spawns for each input file an own worker process that is only responsible for that file. We limit the amount of concurrently spawned processes by the virtual processors available (like in the process pool from Python). How this pool is exactly implemented, we discuss in the next section.

3.3.6 Process Pool

The process pool starts with the graph of the pipeline and a list of input files. Each input file gets assigned to an own sub-process. We make a sub-process by using the process class from the multiprocessing module. We store them in a queue and once the pool starts, they get taken out of it one after another. In order to limit the maximum amount of concurrent worker processes, it was necessary to implement a sort of limiting mechanism. This mechanism is a class encapsulating an event object and a value object. The value object is a wrapper to a value allocated in shared memory, which means every sub-process has access to it. The event object on the other hand can be set or cleared and therefore be used as synchronization mechanism. After we spawn the initial amount of sub-processes, we set the event and it prevents the process pool from spawning new ones. Once a worker process terminates, the shared value decreases and the event gets cleared allowing the pool to spawn a new process again. In addition, the user can set waiting points in the pipeline. This means that before continuing at a certain point, all processes must reach it. We achieve this behavior through recursively creating a new process pool with the remaining steps. Using a separate logging thread, we can do the logging of the events from each worker process. The reason for that, is the fact that the SQLite wrapper for Python does not allow an opened connection to be shared between multiple threads or processes. The logging thread shares a queue with every sub-process. Once one of them makes a call, we put the logs into the queue where the thread takes it and writes them to the database [Pytc].

Optimization of Python Calls

One big problem, which occurred during development, was when it comes to using the process pool on Windows. Every program call a worker process from the pool makes creates a new sub-process of its own in order to run any arbitrary binary. However, this massively slows down the whole application because of the following reason:

The process object from the multiprocessing module in Python supports multiple ways of spawning a new process. There is the "fork" method, which forks the main process and the child process inherits everything from its parent. This method is rather fast, but the downside is that it only works under UNIX systems. The other method, which is the only one available under Windows, is "spawn". It creates a new instance of the Python interpreter, giving it the necessary resources to work on its own. The big issue with this method is that it is very slow [Pytc].

In order to fix this problem, we made the following optimization. The framework in general, supports calls to any arbitrary program that we need for processing the pipeline. However, when we make a call to another Python script, it is unnecessary to spawn a new Python interpreter. The current interpreter can dynamically load the script as a module and run its main method. With that optimization, no new sub-processes need to be spawned when calling other Python scripts. Since the dataset generation pipeline consists almost entirely of such scripts, we can achieve a huge speed gain especially under Windows.

3.3.7 Dynamic Module Loading

The importlib [Pyta] module from Python provides an implementation of the import statement. However, it also provides functionality to write custom importers and to realize dynamic module loading during runtime. We use the following code in the framework to dynamically call external Python code without spawning a separate sub-process:

```python
called_module_spec = spec_from_file_location(name, location)
if called_module_spec is not None:
  called_module = module_from_spec(called_module_spec)
  called_module_spec.loader.exec_module(called_module)
  called_module.main(args[2:])
```

The code works as followed:

1. It first loads the module given the actual name and the location in the file system using the *spec_from_file_location* function. This creates a *ModuleSpec* object.

2. With the *module_from_spec* function, we create the actual module object and return it.

3. In the final step, the *exec_module* function from the loader, which comes with the *ModuleSpec* object, executes the module.

4. However in order to run actual code inside the module, a function must be called. We do this in the last line that calls the main function of the module with some command line arguments if available.

3.3.8 Logging

In order to receive sufficient information on what happens during the execution of the pipeline, we do logging in two ways. There is log file for any kind of unexpected behavior and exceptions that occur during runtime. It contains a timestamp, an error message and a stack trace to see where we exactly found the exception.

In addition, we use a simple SQLite 3 database to track information about the executed edges and program calls that we make. The structure can be seen in the ER model 3.2. The user is now able to e.g. see which information programs would otherwise print to the console. This output is, during the runtime of the framework, suppressed in order to keep the console clean. This database can be read directly using a program able to read SQLite (e.g. DB Browser for SQLite[6]) or otherwise use the interactive shell of the framework. We provide the following logging functionality trough it:

- Getting the status code and console output of the call for a given file it created.
- Writing the console outputs of all calls of the last run to an own log file.
- Get an overview of the edges of the last run. This overview contains which edges we executed, how many calls we made per edge and how many of them were successful.
- Get a list of calls, which lead to the creation of one given file.

[6]`https://sqlitebrowser.org/`

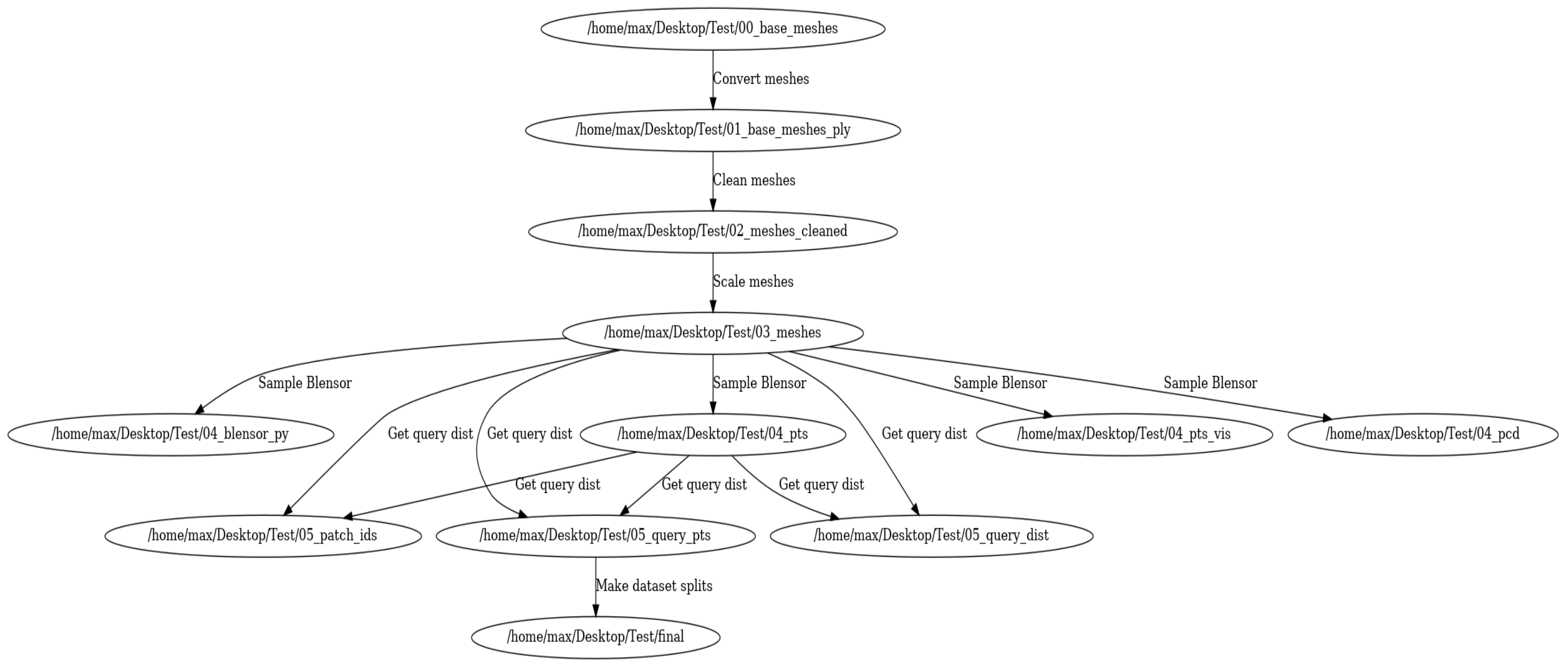

/home/max/Desktop/Test/00_base_meshes
Convert meshes
/home/max/Desktop/Test/01_base_meshes_ply
Clean meshes
/home/max/Desktop/Test/02_meshes_cleaned
Scale meshes
/home/max/Desktop/Test/03_meshes
Sample Blensor
Sample Blensor
Sample Blensor
Sample Blensor
/home/max/Desktop/Test/04_blensor_py
Get query dist
Get query dist
/home/max/Desktop/Test/04_pts
Get query dist
/home/max/Desktop/Test/04_pts_vis
/home/max/Desktop/Test/04_pcd
Get query dist
Get query dist
Get query dist
/home/max/Desktop/Test/05_patch_ids
/home/max/Desktop/Test/05_query_pts
/home/max/Desktop/Test/05_query_dist
Make dataset splits
/home/max/Desktop/Test/final

Figure 3.1: Example pipeline graph. Each node represents a directory in the file system. We connect the nodes using directed edges, which correspond to a program call. These calls process files from the folder at the beginning of the edge and put the resulting files in the folder at the end of the edge.

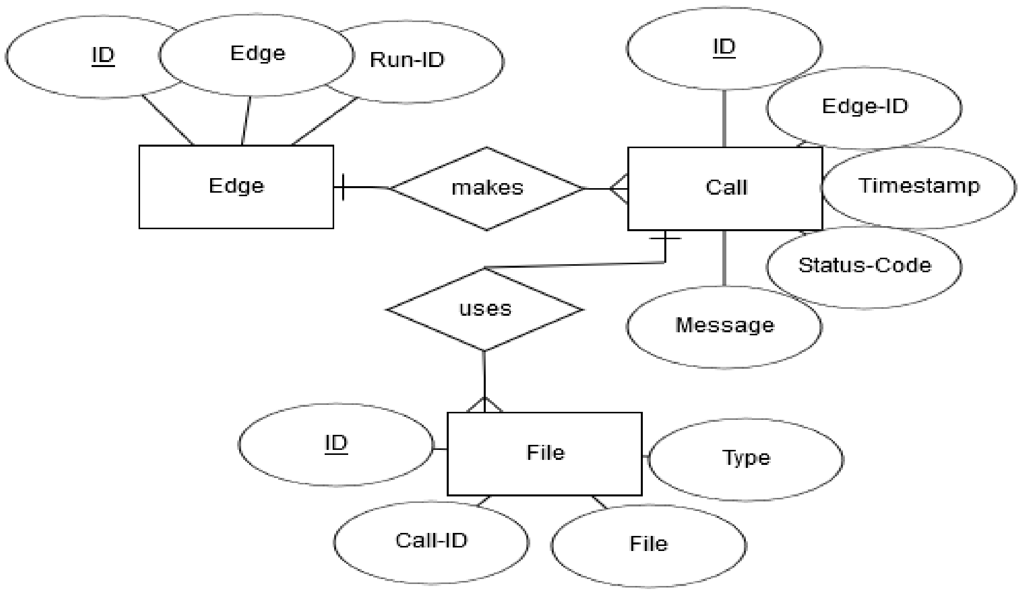


Figure 3.2: ER model using Crow's Foot Notation. Entities are rectangles and attributes are in ellipses. If we underline an attribute, it is the primary key of that entity. Entities connect to one another with relations (diamond shape). Relations in this diagram are all 1:n with the 1-side represented as a bar and the n-side as a "crow's foot".

CHAPTER 4

Experiment

This chapter explains the experiment in order to see, if the framework performs better using the self-developed process pool instead of running everything sequential.

4.1 Hypothesis

We did the development of the framework in a kind of iterative process, where we added functionality step by step with working prototypes in between iteration cycles. Thus, multiple versions of the framework, which use different ways of processing the files, emerged. There are a total of 3 of such processing methods named "execution models" that are:

- Sequential: Each file gets processed in sequence before continuing to the next step of the pipeline.
- Simple Process Pool: Each step gets processed using the available process pool of Python (multiprocessing module).
- Global Process Pool: An own implementation of a process pool. The idea is to process a file in its own sub-process through every step of the pipeline (more details see Section 3.3.6).

Now we make a benchmark by measuring the time of execution for each of these execution models using a different number of files every time. The number of files used are 1, 3, 10, 50, 100 and 500. We measure the wall-clock time using the perf_counter() [1] function of Python's time module. We call the function before and after running the framework.

[1] `https://docs.python.org/3/library/time.html#time.perf_counter`

The difference between both returned times is the execution time. We do the whole benchmark process under a Windows 10 system with an Intel i5-6600K CPU, which has a total of 4 processors (physical and virtual) and 3.50 GHz clock speed.

The hypothesis that we made here is that both execution models with a kind of process pool perform better on average than the sequential model. We made this assumption because the process pools parallelize the execution and therefore utilize multiple cores of the CPU. However, it cannot be said for sure, because spawning new sub-processes every time leads to overhead that can potentially slow down the whole process.

4.1.1 Results

The results of the benchmark can be seen in the Table 4.1 and in the Graph 4.1. The first observation that can be made is that, when only processing one file, the sequential method is the fastest. This makes sense, because the process pools do not spawn multiple processes and therefore only produce additional overhead. Once we use more than one file, the process pools start to gain an advantage in speed compared to sequential processing especially with a larger number of files. Thus, the hypothesis seems to be true that the process pools perform in general better. This makes sense because of the complete parallelization achieved using the process pool. Instead of one file, multiple ones can be processed at the same time.

Another observation that can be made is that the process pool developed during this thesis performs at first slightly better than the one Python is using. However, when processing 500 files, the Global Process Pool is about 20 seconds slower than the other one. This can be the case because the overhead is bigger than some waiting times at a certain number of files. Further profiling and analysis is be necessary to determine the exact cause. The fact that this pool prioritizes producing output files as fast as possible, makes this execution model nonetheless the most convenient choice between all of them.

Number of files	Sequential	Simple Process Pool	Global Process Pool
1	13.7s	23.5s	14.9s
3	58.9s	39.0s	29.7s
10	217.8s	83.8s	79.4s
50	1236.0s	385.2s	377.5s
100	2532.9s	781.0s	764.3s
500	11827.3s	3403.6s	3425.9s

Table 4.1: Benchmark results. Shows execution times for given number of files and for each execution model. The sequential model processes each edge of the pipeline one after another. Inside each edge we process each file sequentially as well. The Simple Process Pool is the available process pool from Python. We use it to split the work of each edge among multiple processors. Each processor becomes a part of the files which need to be processed. After an edge is complete we spawn a new processor pool for the next one. The Global Process Pool is a custom developed process pool. It spawns individual processes for each input file and works them through the whole pipeline. After a process is finished it takes the next input file. More details on this process pool can be found in Chapter 3.3.6.

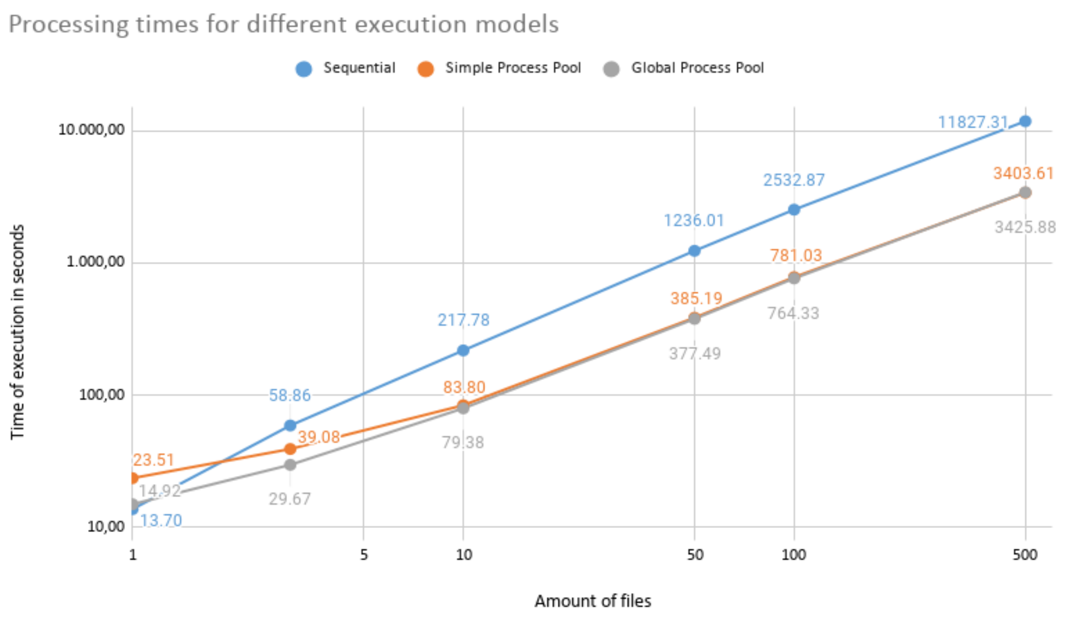


Figure 4.1: Visual representation of benchmark results. The sequential model performs the slowest except when we only processing one file. Both process pools perform significantly better than the sequential model. In addition, the Global Process Pool is a little bit faster than the Simple Process Pool until 500 files. When we process 500 files, the Global Process Pool becomes about 20 seconds slower. This can be the case because of some overhead that only becomes visible at a certain number of files.

CHAPTER 5

Conclusion & Future work

This thesis implements a framework, with the main purpose of generating datasets for surface reconstruction. It uses a custom process pool, which utilizes multi-processing and a depth-first approach. The big advantage here is that if the user wants to process many files, he will quickly get the final results for the first few inputs, before moving on to the next input files. This allows the user to already make use of some results while the framework is still working in the background.

The results of the experiment (Chapter 4.1.1) show that using some kind of process pool achieves faster results than processing every file in sequence on a single thread/process. Furthermore, the approach with the custom process pool seems to get lower total processing times compared to using Python's process pool until a certain point. However, in Chapter 4.1.1 we made the observation that with 500 files, the Global Process Pool starts to fall behind the Simple Process Pool. Further analysis is needed to find out the exact cause for such behaviour.

5.1 Future work

A next interesting step is to look at possibilities to further increase the speed of the framework. This can be achieved by combining the framework with a compute cluster. Many available processors directly influence the speed of the process pool, because more files can be processed at the same time. In addition, frameworks like Hadoop [1] can also be useful, though the typical MapReduce paradigm cannot be directly applied to the processing pipeline.

[1] `https://hadoop.apache.org/`

Bibliography

[CFG$^+$15] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *CoRR*, abs/1512.03012, 2015.

[EGO$^+$20] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Michael Wimmer, and Niloy J. Mitra. Points2Surf: Learning implicit surfaces from point clouds. *arXiv preprint arXiv:2007.10453*, 2020.

[GKUP11] Michael Gschwandtner, Roland Kwitt, Andreas Uhl, and Wolfgang Pree. Blensor: Blender sensor simulation toolbox. In *International Symposium on Visual Computing*, pages 199–208. Springer, 2011.

[HM05] S. V. Hashemian and F. Mavaddat. A graph-based approach to web services composition. In *The 2005 Symposium on Applications and the Internet*, pages 183–189, 2005.

[KMJ$^+$19] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[LLVT03] Thomas Lewiner, Hélio Lopes, Antônio Wilson Vieira, and Geovan Tavares. Efficient implementation of marching cubes cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, december 2003.

[Pyta] Python Software Foundation. Python 3 Documentation - importlib. `https://docs.python.org/3/library/importlib.html`. [Online; accessed 09-July-2020].

[Pytb] Python Software Foundation. Python 3 Documentation - Initialization, Finalization, and Threads. `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`. [Online; accessed 11-July-2020].

[Pytc] Python Software Foundation. Python 3 Documentation - multiprocessing. `https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods`. [Online; accessed 27-June-2020].

Appendix

XML schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning" vc:minVersion="1.1">
  <xsd:element name="edge-set">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="edge" minOccurs="0" maxOccurs="unbounded"
        type="edgeType"/>
      </xsd:sequence>
      <xsd:attribute name="version" type="versionNumber" default="0.1"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="edgeType">
    <xsd:sequence>
      <xsd:element name="name" minOccurs="1" maxOccurs="1"
      type="xsd:string"/>
      <xsd:element name="description" minOccurs="1" maxOccurs="1"
      type="xsd:string"/>
      <xsd:element name="inputDir" minOccurs="1" maxOccurs="unbounded"
      type="xsd:string"/>
      <xsd:element name="outputDir" minOccurs="1" maxOccurs="unbounded"
      type="xsd:string"/>
      <xsd:element name="call" minOccurs="1" maxOccurs="1"
      type="xsd:string"/>
      <xsd:element name="parameters" minOccurs="1" maxOccurs="1"
      type="parametersType"/>
    </xsd:sequence>
    <xsd:attribute name="type" type="edgeCategory" default="regular"/>
    <xsd:attribute name="require_complete" type="xsd:boolean"/>
  </xsd:complexType>
  <xsd:complexType name="parametersType">
    <xsd:all>
      <xsd:element name="option" minOccurs="0" maxOccurs="unbounded"
       type="optionType"/>
      <xsd:element name="argument" minOccurs="0" maxOccurs="unbounded"
       type="argumentType"/>
    </xsd:all>
  </xsd:complexType>

  <xsd:complexType name="optionType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="typeType"/>
    <xsd:attribute name="value" type="xsd:string" default=""/>
    <xsd:assert
     test="(@type␣and␣not(@value))␣or␣(not(@type)␣and␣@value)
␣␣␣␣or␣(not(@type)␣and␣not(@value))"/>
  </xsd:complexType>

  <xsd:complexType name="argumentType">
    <xsd:attribute name="type" type="typeType"/>
    <xsd:attribute name="value" type="xsd:string" default=""/>
    <xsd:assert
     test="(@type␣and␣not(@value))␣or␣(not(@type)␣and␣@value)"/>
  </xsd:complexType>

  <xsd:simpleType name="typeType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="input"/>
      <xsd:enumeration value="output"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="versionNumber">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9].[0-9]"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="edgeCategory">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="regular"/>
      <xsd:enumeration value="unique"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

This XML schema describes how a valid configuration file needs to look like. It itself is written in XML and defines which elements and in which order they need to occur to verify correctly. The "xsd" prefix here is a specific namespace used to identify every element that comes from XML schema itself. The name of that namespace can be arbitrarily chosen. However, we use "xs" or "xsd" most of the time.

An "xsd:element" represents an XML tag and how it has to look like. It can define what attributes it can or must have. In addition, a type for the element can be chosen. There are, on the one hand, predefined types like strings, integers or booleans. These types can be further restricted and/or extended using the "simpleType" element. Such "simpleType" can make sense e.g. if you want a string to have a specific pattern or can only be taken from a fixed set of possibilites. However, these types are only useful if the element should only contain text or nothing. We use the "complexType" element to define a type in more detail. It can say e.g. that certain elements should contain a specific number of elements from another type. Further refinements, like demanding a certain order for the elements to occur, allow the creation of very complex types.

The introduction of assertions in XML schema 1.1 allows even more control. They can be used to check and compare certain values or attributes. In this example we use an assertion to define the complex type named "argumentType". An element that has this type must either have an attribute named "type" or an attribute named "value". It does not allow to have none or both attributes like an XOR.