VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Configuration Interface for the Tool ESPRESO

Konfigurační rozhraní pro nástroj ESPRESO

2018 Tomáš Panoc

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student:	**Bc. Tomáš Panoc**
Study Programme:	N2647 Information and Communication Technology
Study Branch:	2612T025 Computer Science and Technology
Title:	Configuration Interface for the Tool ESPRESO Konfigurační rozhraní pro nástroj ESPRESO
The thesis language:	English

Description:

Main goal of the master's thesis is to create a graphical user interface (GUI) for setting and controlling computations based on finite element method in the numerical library ESPRESO (ExaScale PaRallel FETI Solver). ESPRESO is developed at IT4Innovations National Supercomputing Center. Focus of the development team is to create a highly efficient parallel solver for structural mechanic problems. Solver contains several FETI based algorithms suitable for parallel machines, even with thousands of cores.
The thesis objectives could be summarized in following points:
1. Familiarization with the ESPRESO library, its structure, possibilities of input definition and types of output.
2. Analysis of available tools for GUI development with a support of 3D rendering. Chosen solution should meet the requirements defined by ESPRESO development team, e. g. open source methodology.
3. Design of a GUI structure. The ease of a future extension with respect to the development of ESPRESO should be considered.
4. Implementation of chosen solution.
5. Testing and verification of final solution on examples of heat transfer computations.

References:

[1] Riha, L., Brzobohaty, T., Markopoulos, A.: Hybrid parallelization of the Total FETI solver, Advances in Engineering Software, pp. -, 2016, ISSN: 0965-9978.
[2] ESPRESO: http://espreso.it4i.cz/

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Marek Běhálek, Ph.D.**

Date of issue: 01.09.2017

Date of submission: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
Head of Department

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, 30th April 2018 .....

I would like to thank my supervisor Ing. Marek Běhálek, Ph.D. for all his help, advises and numerous consultations. I am also grateful to Ing. Ondřej Meca for hundreds of answered questions about ESPRESO. I shall be forever indebted to my parents and family for their unshakeable faith and continued support. Thank you.

This thesis was supported by student grant SP2018/159 "Hardware acceleration of matrix assembler and GUI development of ESPRESO library", VŠB - Technical University of Ostrava, Czech Republic.

Abstrakt

Tématem této práce je vytvoření grafického konfiguračního nástroje pro řešič ESPRESO, který nahrazuje ruční psaní konfiguračních souborů ve formátu ECF. Text práce začíná představením řešiče a vychozího způsobu zadávání vstupních dat. Na tuto část navazuje analýza požadavků, které by měl konfigurační nástroj splňovat, přehled existujicích řešení, a knihoven pro vývoj grafických rozhraní. Následně jsou čtenáři seznámeni s návrhem a vývojem výsledné aplikace. Na závěr je funkčnost aplikace demonstrována na jednoduchém příkladu výpočtu přenosu tepla.

Klíčová slova: grafické uživatelské rozhraní, ESPRESO, konfigurační nástroj, Qt, OpenGL, hybridní paralelizace

Abstract

This thesis deals with the development of a graphical configuration tool for the ESPRESO solver which eliminates the necessity of writing configuration files (ECF) manually. The text starts with the introduction to ESPRESO and its default method for the input definition. This part is followed by the analysis of requirements, that the configuration tool should meet, existing configuration tools, and libraries for the graphical user interface development. Then, readers get to know the design and the development of the final application. Finally, the tool functionality is demonstrated on a practical example of heat transfer computation.

Key Words: graphical user interface, ESPRESO, configuration tool, Qt, OpenGL, hybrid parallelization

Contents

List of symbols and abbreviations

2D	–	Two-Dimensional
3D	–	Three-Dimensional
API	–	Application Programming Interface
BEM	–	Boundary Element Method
CAD	–	Computer Aided Design
CD	–	Compact Disc
CPU	–	Central Processing Unit
ECF	–	ESPRESO Configuration File
EDT	–	ESPRESO Development Team
ESPRESO	–	ExaScale PaRallel FETI SOlver
FEM	–	Finite Element Method
FETI	–	Finite Element Tearing and Interconnect
GCC	–	GNU Compiler Collection
GLSL	–	OpenGL Shading Language
GPU	–	Graphic Processing Unit
GTK+	–	Gimp ToolKit
GUI	–	Graphical User Interface
HPC	–	High Performance Computing
IDE	–	Integrated Development Environment
JPEG	–	Joint Photographic Experts Group
JSON	–	JavaScript Object Notation
MacOS	–	Macintosh Operating System
MIC	–	Many Integrated Core
MPI	–	Message Passing Interface
OpenMP	–	Open Multi-Processing
OpenGL	–	Open Graphics Library
PDF	–	Portable Document Format
QML	–	Qt Modeling Language
RGB	–	Red Green Blue
SQL	–	Structured Query Language
XML	–	eXtensible Markup Language

List of Figures

List of Tables

Listings

1 Introduction

ExaScale PaRallel FETI Solver (ESPRESO) is a parallel framework for computing structural mechanics tasks. Currently, the ESPRESO solver features a module for heat transfer problems. To set up a computational task for ESPRESO, one has to use a special configuration file. Since the number of all possible parameters, that ESPRESO enables to specify, is relatively high (hundreds), it is quite difficult to create the configuration file without a list of all parameters with their meaning. The situation is even more complicated, because some of the parameters can be specified only when some other parameter is present, otherwise, ESPRESO would not take them into consideration. Naturally, one starts here thinking whether it would be possible to provide a better way to configure the solver, and also help new users to quickly start using ESPRESO.

One way to simplify the configuration process might be the introduction of a graphical user interface (GUI). It would serve as a wizard, which would help users with the configuration. Development of such tool is the main goal of this thesis.

In Chapter 2, a brief introduction to the ESPRESO framework together with the configuration files are presented. Chapter 3 analyses requirements, that the outcome of this thesis should meet. Also, a few existing graphical configuration tools are examined there, followed by the overview of libraries enabling programming of GUIs. Functional requirements of GUI are discussed in detail in Chapter 4. Chapters 5, 6, 7 and 8 describe the development process of the final application step by step. The functionality of the application is demonstrated on a simple example in Chapter 9.

2 ESPRESO

ESPRESO is a massively parallel framework based on the Finite Element Method for Engineering Application. The main objective of the ESPRESO framework development team is to create a robust open-source package applicable for a wide range of complex engineering simulations in areas such as mechanical engineering, civil engineering, biomechanics, and energy industry.

The ESPRESO framework consists of several logical units for each phase of a specific solution task. These units include input data pre-processing, computational mesh processing, FEM/BEM objects builders, solution of the specific physical problem using a massively parallel sparse linear solver, and finally output data preparation for visualisation of the results. The ESPRESO project started as an open-source ExaScale PaRallel FETI (a numerical method - finite element tearing and interconnect) Solver for problems of engineering mechanics developed at the IT4Innovations center within the FP7 EXA2CT project. At the end of the project, ESPRESO contains several FETI based domain decomposition algorithms including the Hybrid Total FETI method suitable for parallel machines with tens or hundreds of thousands of cores. The solver is based on a highly efficient communication layer on top of pure MPI. Its scalability has been tested up to 18,000 compute nodes of the ORNL Titan supercomputer. ESPRESO supports GPUs and MICs acceleration. By dynamically tuning several key hardware parameters ESPRESO can save over 20% of its energy consumption. It can also be accelerated by both GPU or Intel Xeon Phi accelerators to achieve up to 5x higher performance improvements over general purpose CPUs. With all these properties, it has real potential to efficiently utilise future exascale systems where both performance and energy efficiency matter.

The ESPRESO team is currently focusing on development of a complex framework, which will be ready to provide a whole toolchain for solving challenging engineering problems (e.g., heat transfer, optimisation in structural mechanics, air and water pollution transport, topology optimisation, etc.). In Q2 2018, a module for solution of the heat transfer problems via the ESPRESO framework was completed. A new module for structural mechanics including contact problems, module for ultrasound propagation, module for topology optimisation, and module for design of experiment and shape optimisation based on mesh morphing will be developed.

ESPRESO offers hundreds of parameters and options that influence a computation. Also, it features several numerical methods used in different stages of computation. We are not going to clarify how these methods work and cooperate, because they are not essential to this thesis. For us, it will be more beneficial to show some of parameters for computation configuration, because on the ground of their amount, one understands why GUI is necessary.

2.1 Input

Configuration of ESPRESO is performed by configuration files which use own format ECF (**.ecf*), i.e., ESPRESO Configuration File. The ESPRESO solver expects a path to ECF as the input argument.

ECF is a text file, the structure of which is simple, and one might find it similar to JSON, but they differ. Basic configuration file element is called *parameter*. There are two types of parameters. First, the *value* is a key-value pair of strings finished with a semicolon. Second, the *object* starts with a key string, and its content is enclosed by curly brackets. The content can be composed of a combination of objects and values. The format supports comments that begin with hash symbol #. A demonstration of the ECF structure could be seen in Listing 1. Nevertheless, this example shows a general composition of the ECF format and will not work with ESPRESO. ESPRESO defines an exact list of values and objects, which could or should be used according to the given context. At the time of writing this thesis, no complete list was available as this part of ESPRESO was still in development. On the other hand, this work is focused on the development of a tool, which will allow us to avoid excessive studying of endless lists of parameters.

```
# Here lies a comment...

# An example of value
VALUE_KEY_1 1024;

# An example of object
MY_OBJECT {
    MY_VALUE1 1;
    MY_VALUE2 HELLO_WORLD;

    INNER_OBJECT1 {
        MY_VALUE1 TRUE;
    }
}
```

Listing 1: Example of the ECF format structure

2.2 ECF structure

Let us inspect a predefined structure of ECF values and objects. We are not going to examine every parameter that ESPRESO supports because it is not necessary as we will see in the following chapters. For now, we just need to know a few objects and values which are essential. They will help us to understand how ESPRESO works.

Classic formats like XML or JSON usually start with some kind of a root element. That is not the case of ECF. There is no root object which we should specify. We could imagine that ECF itself forms the root.

2.2.1 Mesh

ESPRESO requires information about a 2D/3D model for which the computations are performed. Thus, a source of this data has to be specified. There is a value parameter called *INPUT* which represents an input format of the model data. ESPRESO currently supports:

- ESPRESO mesh generator (*GENERATOR*).
- ESPRESO binary format (*ESDATA*).
- Ansys Workbench format (*WORKBENCH*).
- OpenFOAM format (*OPENFOAM*).

For all of these options, the solver expects that an object with a key, which is equal to the content of the *INPUT* parameter, exists in ECF. We will not go through the details of every model data format, but we can briefly introduce the formats.

Mesh generator could be used to specify how the mesh should look like, and ESPRESO generates it accordingly. In the language of ECF, there would be an object *GENERATOR*, which would contain predefined parameters describing the mesh. The rest of the formats match the existing file formats, which include the generated mesh already. The corresponding ECF object would include a path to such file.

The mesh provides information about the geometry and may divide the model into regions for which one is able to define another properties and conditions. ESPRESO enables conversion of all mentioned formats into its own *ESDATA* binary format.

2.2.2 Physics

The mesh defines what geometry we will be computing with, while physics determines what computations the solver will perform. The procedure is identical as in the case of the mesh. There is a *PHYSICS* value parameter, which currently supports the following physics:

- 2D or 3D heat transfer (*HEAT_TRANSFER_2D* and *HEAT_TRANSFER_3D*).
- 2D or 3D structural mechanics (*STRUCTURAL_MECHANICS_2D* and *STRUCTURAL_MECHANICS_3D*).

The settings of physics should be specified in the corresponding object named after the value in *PHYSICS*. The object is composed of other objects which will be inspected.

First, materials (*MATERIALS*) form a set of objects in which one may define a material, especially its physical properties. The properties differ according to the selected physics. Hence, depending on physics, different parameters are expected to be present in a material object. The material could be attached to a region (e.g., a steel would be assigned to a region representing a car brake plate).

Second, users may divide computations into a sequence of steps where a result of the first step serves as an input for the second step, and the result of the second step goes to the third step, and so on. These steps are called *load steps* in ESPRESO, and the value parameter *LOAD_STEPS* tells the solver how many load steps will be present. Inside a *LOAD_STEPS_SETTINGS* object, a configuration of every load step lies. Within the load step users assign additional properties (i.e., *boundary conditions*) to regions and configure the solver. Like in the case of the material parameters, the load step parameters (especially the properties) depend on the selected physics.

2.2.3 Output configuration

The last interesting parameter is an *OUTPUT* object which defines what data will be gathered during computations, and where the solver will store them.

When we put all previous parameters together we could assemble ECF as it is shown in Listing 2.

```
# Comment in an imaginary root object
INPUT  GENERATOR;
GENERATOR {
    # Mesh generator details...
    # Let us assume that we defined here region REGION_1
}
PHYSICS HEAT_TRANSFER_3D;
HEAT_TRANSFER_3D {
    MATERIALS {
        # Material definition lies here...
        # Let us assume that we defined here material MATERIAL_1
    }
    # Assignment of materials to regions
    MATERIAL_SET {
        REGION_1   MATERIAL_1;
    }
    LOAD_STEPS 2;
    LOAD_STEPS_SETTINGS {
        1 {
            # Configuration of loadstep 1...
        }
        2 {
            # Configuration of loadstep 2...
        }
```

```
    }
    # Here could be other physics parameters...
}
OUTPUT {
    # Configuration of output...
}
```

Listing 2: Simplified example of ECF

2.3 Output

ESPRESO's input is composed of the mesh and ECF which only extends the mesh by the introduction of materials, physics, properties, etc. The output of ESPRESO is also composed of a visual part and its description. For example, if we compute a heat transfer the output will contain information about the heat in all parts of mesh. ESPRESO packs this data into a proprietary EnSight format which can be opened by a software of the same name. Nevertheless, this format is supported by open source software including ParaView.

Apart from the computational results, ESPRESO enables gathering of statistical data during execution. It can monitor changes of physical properties in selected regions and provide information about maximum value, minimum value, or average.

3 Analysis

In the following sections, we are going to investigate basic demands expected from the final product (GUI) by the authors of ESPRESO. The following paragraphs give insight into software packages similar to ESPRESO and their solutions. On the basis of the analysis of demands and the existing solutions, Chapter 3.3 examines possible programming tools suitable for development of the final product.

3.1 Requirements

An essential requirement is the development of a configuration tool which substitutes manual typing of ESPRESO configuration files. The tool should provide a graphical user interface that enables users to set all necessary parameters in order to produce a valid configuration file. GUI should be intuitive and interact with users, e.g., respond to an attempt of invalid input appropriately. Since ESPRESO performs computations for some input model built from a 2D/3D mesh, GUI should visualise the mesh.

3.1.1 General constraints

Since ESPRESO represents a kind of framework, integrating GUI into it is limited by the framework model and philosophy. Let us have a look at a few constraints and recommendations implied by the usage of ESPRESO. The following list also includes the demands which the ESPRESO development team (EDT) places on GUI.

1. ESPRESO follows the open source philosophy, thus its source code is available to everyone and all the third party libraries are open too. GUI should not break this model.

2. Main development language is C++. ESPRESO already implements a parser for the ECF files featuring functions for reading, writing, and validation. Hence, GUI written in C++ could directly use it, and moreover, the immediate connection with ESPRESO brings a possibility to execute computations instantly from GUI by calling the solver.

3. GUI should support multiple platforms. Despite the fact that ESPRESO runs only on the Linux systems, GUI should be ready to work on all common desktop platforms, i.e., Windows, macOS, and Linux. This requirement collides with the previous idea of direct connection with ESPRESO. Nevertheless, from the EDT point of view it would be acceptable. Thus, the part of ESPRESO with GUI would be platform independent but the whole ESPRESO would not.

4. ESPRESO is developed for application in the HPC area, i.e., supercomputers and clusters. GUI should use sources (e.g., third-party libraries) that are commonly found in such systems.

3.1.2 Use case

The main goal of GUI is to produce correct ECF files. A basic scenario leading to a well-formed configuration file looks as follows:

1. Import of input file with mesh.
2. Mesh 3D visualisation.
3. Selection of physics.
4. Definition of materials.
5. Configuration of physics.
6. Configuration of loadsteps.
7. Configuration of output.
8. Validation of configuration.
9. Export of configuration file.

Steps 3 and 4 may be swapped since a material is a set of properties from which a subset is chosen according to the selected physics. The points above form a core use case. Other use cases would just extend individual points in the core one. They are not important for now.

3.2 Existing solutions

We are going to introduce a few open source software packs which are similar to the ESPRESO solver and already feature GUI. During the reading of the following sections, you may find Appendix B helpful. It contains figures of all tools.

3.2.1 Elmer

Elmer is an open source multiphysical simulation software developed by CSC - IT Center For Science in Finland. It supports a variety of physics and mathematical models including heat transfer, fluid flow, elasticity, etc. The software structure is divided into several mutually cooperating independent blocks. [2]

ElmerSolver, which is the core computational part, is similar to ESPRESO. Both perform the computations with input parameters including the given model with chosen properties.[4]

ElmerGUI is GUI for the solver configuration and represents exactly that kind of application which is introduced for ESPRESO by this thesis. *ElmerGUI* features the following:

- Import of several mesh file types and visualisation of mesh.
- Mesh is divided into *bodies* which have the same meaning as ESPRESO regions.

- Definition of *equations* which are equivalent to selection and configuration of physics in ESPRESO.
- Definition of materials and conditions (e.g., initial, boundary, body force, etc.). Both can be attached to a body.
- Saving configuration to a drive.
- Running the solver.
- Post processing with *ElmerPost*, i.e., another GUI tool, or via a built-in console. There are consoles for Python and ECMAScript. Both contain ready-made API classes leading to visualisation of results. [3]

ElmerGUI is not compatible with ESPRESO since both use different mesh and configuration formats.

3.2.2 CalculiX

CalculiX is another open source tool for multiphysical simulations developed by a team from MTU Aero Engines in Munich, Germany. It consists of two main parts, *CalculiX GraphiX*, and *CalculiX CrunchiX*. The first one is GUI and the second one is a solver. In contrary to *ElmerGUI*, *GraphiX* does not enable the solver configuration. It is just a visualisation tool which can draw a mesh in different view modes, perform animations, or export the model to standard picture formats like JPEG. Also, it can display the results of a computation. On the other hand, there exists a third party software CalculiX Launcher for the *CrunchiX* configuration. [5, 6, 7]

3.2.3 SALOME

SALOME is the last software that we will concentrate on. It differs from the previous two, because it is not a package of programs, but it is a standalone application specialised for pre-processing and post processing. Pre-processing includes geometry construction and mesh generation. It features CAD 3D modelling tools, which users are able to construct any geometry completely from scratch with. Post-processing enables to display the computed results. [8]

SALOME does not include any solver. Nevertheless, its structure is modular, meaning that the whole application is separated into components such as the geometry creator, the mesh generator or the post processing visualisation tool. New components can be added by anyone. Thus, developers could build a module that would add a functionality for setting up a specific solver inclusive of physics, materials, properties, etc. [8]

There exists a software package including SALOME and a module providing bindings to the Code_Aster solver. It is called Salome-Meca. The solver module offers standard options such as physics settings, attaching boundary conditions to regions, and many more. All these options have slightly different names in Salome-Meca, since Code_Aster uses own work flow. After

every important parameter is set, users may run a computation task from GUI, and monitor the progress through built-in console. When the task is done, output data can be visualised immediately. [9, 10]

3.2.4 Summary of solutions

From the perspective of this thesis, the least interesting solution is provided by CalculiX since its *GraphiX* GUI offers the mesh visualisation only. The solver configuration is missing.

As a beginner in the field of multiphysics solvers with some soft knowledge of ESPRESO and its purpose, I have found *ElmerGUI* to be the most intuitive. The reason is that it uses terminology similar to ESPRESO, and the whole Elmer project has a transparent documentation that is relatively easy to understand. On the other hand, their GUI is more focused on functionality than user experience. All settings are hidden in one top menu. If users want to see, for example, what boundary conditions they set, they have to always open this menu. The main window shows only mesh, and if users use full screen mode there is still a lot of unused space which could be used to display properties, materials, etc.

SALOME looks like a real professional tool. It provides many options and modules, and its derivative Salome-Meca forms a complete multiphysics suite. In my opinion, the entrance knowledge for using the suite should be the biggest among all mentioned tools. SALOME's GUI is really complex, and its documentation is not the best in terms of clear arrangement. However, this is usually a bottleneck of every complex tool. On the other hand, it offers possibility to extend it with own modules which could be taken into consideration as an alternative to the development of GUI for ESPRESO from scratch.

3.3 Libraries and frameworks

In Chapter 3.1.1, we have presented a few points that indicate how the final solution should look like. Let us focus on the points where C++ and the multiple platform support are mentioned. In the field of GUI development, there is no standard tool for C++. C++ standard library does not contain anything like that. Thus, GUIs are commonly developed by using a third party library. Some of them guarantee platform independence, which is important in our case. Also, we need an open source library which can be applied in HPC area.

3.3.1 GTK+

GTK+ is a multi-platform toolkit for GUI development. It is an open source, and one can use it for creating free, or proprietary software. It supports three platforms, i.e., Linux, Windows, and macOS. [11]

Linux systems have usually GTK+ pre-built and available in a packaging system. In contrary, Windows does not offer anything like that, thus in order to develop a GTK+ application, one should use MSYS2 software, which introduces a packaging system called Pacman into Windows

providing the GTK+ installation. Every GTK+ program should be packaged with GTK+ binaries, otherwise, it will not work on Windows since the library is not present in the operating system by default. [12, 13]

Finally, we should not miss macOS support. It requires the greatest effort to make it work. The GTK+ documentation contains plenty of manuals and instructions, which lead to a successful application compilation on macOS. The reason, why the process on the Apple's system is little bit more complicated, is that some of GTK+ modules are not kept updated on this platform. A lot of these problems might be partially avoided by using GTK-OSX tools, that try to simplify the whole process. [14]

HPC clusters Salomon and Anselm maintained by IT4I offer old GTK+ 2.24 (current is 3.22) within their module systems, but cluster users are free to compile what they need.

The development of GUI with GTK+ could be simplified via a third-party GUI builder Glade, which provides a palette of GUI components (e.g., buttons, combo boxes,...). They could be easily dragged and dropped to a program window. The GTK+ library implements a *GtkBuilder* class featuring an ability of loading of an XML document produced by Glade. Glade is mainly developed for Linux. There exist releases for Windows which are usually older than the Linux variant. [15]

3.3.2 wxWidgets

wxWidgets is another library which is an open source, and supports all three platforms. In comparison with GTK+, wxWidgets is not just a library for creating windows, buttons, and things like that. It introduces multi-platform sockets, multithreading, or file management. Hence, we may say that wxWidgets forms a platform independent framework rather than a GUI library. It is interesting that the framework uses GTK+ for Linux applications. For the rest of platforms, it provides own modules different from GTK+. [16]

wxWidgets is not integrated in any operating system that we are interested in. For Linux, there exists an official repository. In the case of Windows, we have to build the library from source, or use prepared dynamic libraries for specific compiler. MacOS requires compilation from source. [17]

Salomon and Anselm clusters do not offer any version of wxWidgets. The only possibility is to compile it one's own.

There is no official IDE for wxWidgets development. Nevertheless, some third party tools exist like wxFormBuilder (GUI designer), wxCrafter (plugin for CodeLite IDE), or wxSmith (plugin for Code::Blocks IDE). The framework accepts XML documents (XRC format) with the defined GUI structure. [18]

3.3.3 Qt

The last library, that we are going to study, is called Qt. It is the most sophisticated tool from all libraries we have come through, because it supports more platforms than the others. Also, it has own official, fully-featured Qt Creator IDE, and one may purchase a license with extended support of Qt team. [19]

Basic variant of Qt is open source with the restriction of distributing applications as open source. That is sufficient for the needs of ESPRESO. If someone wants to develop a proprietary software, they have to purchase the Qt license. On the other hand, the official Qt helpdesk would be provided to such client. Nevertheless, users of the open source license can use official forums with a large community, where one usually receives a help no later than after one day from my personal experience. [19]

Qt forms a complete framework providing an interface for file management, sockets, SQL database management, threading, 3D graphics, and many more. [21]

The framework supports Linux, Windows, macOS, and also Android. Nonetheless, we will stay focused on the first triplet. The compilation, and the development processes are easier than in the case of previous libraries, since users have to download a Qt bundle dedicated for a particular platform. This package contains Qt libraries, compiler, and Qt Creator. After the package installation, user have available a complete development environment. They can use Qt Creator with a built-in GUI designer for code writing and creating GUI. Within the same tool, users may compile the final application. [19, 21, 20]

Salomon and Anselm clusters have pre-installed Qt modules in versions 4.8 and 5.8. Current version available on the official Qt website is 5.10.

3.3.4 Summary of libraries and frameworks

From the previous lines, it is obvious that Qt offers the most comfortable solution. GTK+ requires more effort to be built on multiple platforms, and IT4I clusters contain the old version. In the case of wxWidgets, the situation looks better in terms of platform independence with its system features (e.g., sockets). On the other hand, for all platforms, we should compile the library from sources, if we take into account that the clusters do not offer any module.

It seems that with Qt all the problems would disappear. A relatively new version is present on the clusters. Platform independence should be solved by using their IDE. Moreover, the framework integrates support for 3D graphics, more precisely a universal layer above the OpenGL technology. Hence, GUI for ESPRESO will be developed with Qt.

3.4 Summary of analysis

We came through several sections examining various topics, which should help us imagine, how the final product should look like. Let us summarise it in a few points:

- GUI will be written in C++ as a built-in part of ESPRESO library.

- The graphical part of the final application will use the Qt framework which guarantees the platform independence.
- Direct integration of GUI into ESPRESO will violate the platform independence.
- Alternatively, if we added support for SALOME file formats in ESPRESO and implemented an ESPRESO module for SALOME, we could use SALOME as a GUI layer.

4 Functional requirements

In this section, we will present another set of features, that are expected from GUI by the ESPRESO development team.

4.1 Expressions

ESPRESO works with physics and materials, which contain variety of physical quantities. These quantities could be specified by a value (constant), an expression with a few allowed variables, or a function defined by a table. GUI should extend it, and add a support for a piece-wise function. Together, we will have these options:

1. *Expression* - a mathematical expression with a predefined set of allowed variables. The value (constant) is also an expression.
2. *Table* - a table defining a mathematical function, i.e., a set of couples $(x, f(x))$. See example in Table 1.
3. *Piece-wise function* - a function defined by a set of expressions valid for specific interval. See example in Table 2.

It should be possible to store these structures into a separate abstract container, i.e., a variable. Such variable should be assignable to an unlimited number of physical quantities. Thus, it would reduce data redundancy through all quantities, and make GUI more user friendly, since users would not have to type same thing many times. We may imagine that defining a large table, or a piece-wise function with same content many times could be really time consuming.

4.2 3D model

GUI should be able to display a model of a geometrical object, for which ESPRESO will perform the computations. The object could be a cube, a sphere, a car brake, a valve, a CPU cooler, etc. ESPRESO provides functions for generating a mesh of triangles that forms the object. GUI should use that for the rendering, and enable basic manipulations with the model including rotation and zoom.

Since every object is usually divided into more smaller objects called regions, GUI should colour these regions differently. Also, it is expected that GUI will feature a region hiding, meaning that one can stop the drawing of specific regions. This implies another functionality - a region picking. Users will be provided with something similar to a list of regions, in which they may select a region to hide, or just click on a region causing GUI to highlight the region.

Table 1: An example of a function defined by the table

x	f(x)
0	10
1	20
2	30.6

Table 2: An example of the piece-wise function

Interval	Expression
(-inf, 1>	x^2
(1, 2.5)	$x + 1$
<2.5, inf)	$sin(x)$

4.3 Parallel execution

We have already mentioned the benefit of embedding GUI in ESPRESO library in Chapter 3.1.1. GUI could directly call the solver and start a computation. Since ESPRESO's main feature is parallel execution, GUI should support such execution of the solver. This functionality requires extra attention and will be discussed later.

4.4 Validation

GUI should provide users a way to create a correct ECF file. Thus, it should verify user data and raise an error with an intelligible message about the mistake. In other words, it should not let users to save an invalid ECF file.

5 Design

We have analysed the requirements, the existing solutions, and the tools for the development of GUIs. In this chapter, we will look closely at the input part of ESPRESO, and Qt. On the ground of this knowledge, we will design a GUI architecture.

5.1 Input structure of ESPRESO

In Chapter 2.1, we introduce the basic structure of ESPRESO configuration files, i.e., the ECF format. ESPRESO already contains an interface for working with such files. There is a group of classes which can load ECF, hold its information, and provide functions for editing, validation, and storing. Let us inspect them.

5.1.1 Parameters, values, and objects

We know that ECF is composed of *parameters*. They are represented by an abstract class *ECFParameter* which forms an ancestor of every class standing in for a specific parameter (e.g., materials). *ECFParameter* contains a field of *ECFMetaData* type holding extra information about parameters such as a description, a data type of its value, and many more.

We have already become familiar with *values* and *objects*. In the code, they are implemented as the children of *ECFParameter* - *ECFValue* and *ECFObject*. *ECFParameter* contains $isValue()$ and $isObject()$ methods behaving according to the object type which we call the method on. An *ECFValue* instance would return true in the case of $isValue()$ call and false in the other case. Naturally, *ECFObject* would react in the opposite way. This is one of the main differences between *value* and *object* classes. The second variance relates to class fields. *ECFObject* features a list of parameters which it is composed of. If we return back to Listing 2 we may intuit that, for example, parameter *LOAD_STEPS* would be represented by a child of *ECFValue* whereas *MATERIALS* would be implemented as a descendant of *ECFObject*. Such intuition is correct but classes representing those parameters would not be the direct descendants. ESPRESO uses extra wrapper classes which are not important for now because GUI will work with correctly prepared structure of *ECFValues* and *ECFObjects*. These abstractions of concrete parameters are enough for any work with them.

A complete structure of classes that we have introduced may be seen in Figure 1.

5.1.2 Configuration

In the previous section, we have started with the description of the essential classes which the others inherit from. Now, we will jump to a class that lies at the very top of a class tree. We are going to talk about a class that wraps whole configuration and represents an ECF file.

ECFRoot is a root class of an ECF. Those who are familiar with XML may imagine it as a root element in an XML file. In other words, one can access any parameter included in the ECF

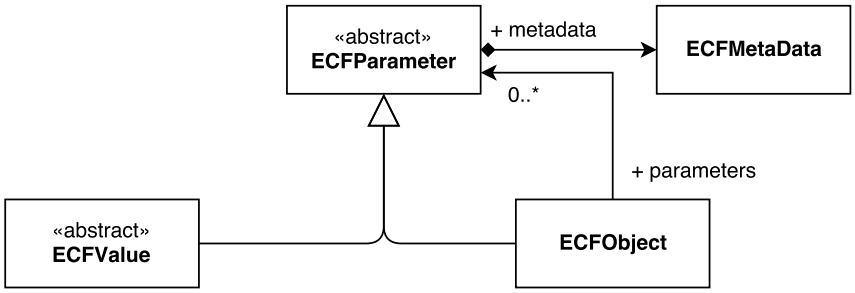

Figure 1: Class diagram of ECF parameters and their structure

file via parameters of *ECFRoot* since the class is nothing more than a descendant of *ECFObject*. Thus, revisiting Listing 2 again, we may guess that *ECFRoot* contains parameters for input, physics, and output. Subsequently, these parameters could be used to reach other parameters.

ECFRoot can directly load an ECF file from a specified path. For saving a configuration, we should use the *ECFReader* class which provides the functionality.

5.1.3 Application in GUI

Usage of *ECFRoot* in GUI is straightforward. The GUI program should scan all parameters and draw them in an appropriate form on the screen. The abstract *ECFParameter* class offers everything we need to get information about each parameter.

In the simplest form, the GUI program could perform a journey through parameters and draw right GUI elements (e.g., a text input box) according to the parameter's data type and group the elements according to the parent *ECFObject* (e.g., all GUI elements of parameters belonging to the *MATERIALS* object would be separated from other elements by a frame with a border). This scenario is great because it does not require any knowledge of concrete parameters. It works only with the abstract *ECFRoot*, *ECFParameter*, *ECFMetaData*, *ECFValue*, and *ECFObject* structures. If we applied this strategy on ECF in Listing 2, we would get a window with structured blocks arranged in a vertical line as it is in Figure 2.

Unfortunately, the proposed strategy leads to bad user experience. We have to distinguish between some of the parameters in order to give users logically arranged GUI, which will follow the use case that we introduced in Chapter 3.1.2. With the previous, scenario the use case is not reasonably practicable as with the knowledge of the five enumerated classes we cannot guarantee which parameter would appear first on the screen. In this case, we would have to rely on ESPRESO that it would put parameters into *ECFRoot* in the correct order. On the other hand, ESPRESO developers might configure a GUI layout from ESPRESO by changing the parameter order. To sum up, the knowledge of some parameters will be necessary.

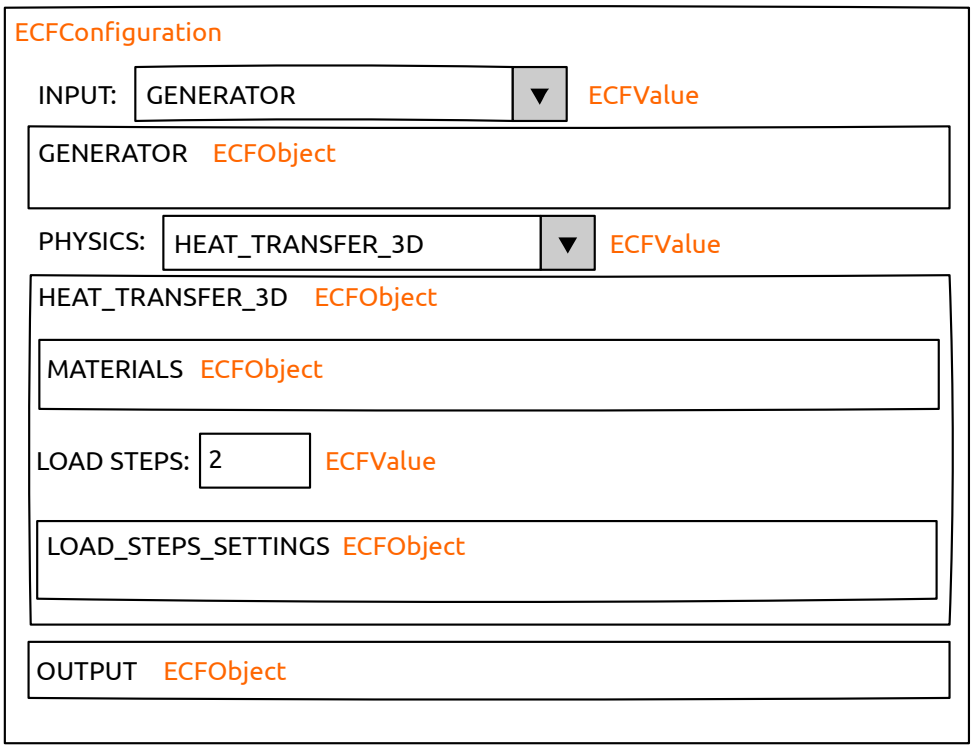

Figure 2: A GUI structure generated by scanning all parameters from Listing 2

5.2 Basic principles of Qt

At the end of previous chapter, we have already come up with an early idea of the strategy for the drawing of graphical elements but we have not showed any details about the elements. Now, it is time to do it. Since we have decided for Qt, we will inspect the capabilities of this library.

Before we start, it is important to mention that this work follows the documentation of Qt 5.10. We will use Qt Widgets, which is a classic Qt library using mainly C++, and it is focused on the development of desktop applications. Alternatively, there is Qt Quick providing a combination of C++ and QML (a programming language similar to JavaScript). Qt Quick features a web-like look, and it is more suitable for mobile devices (e.g., Android systems). [22]

5.2.1 Widgets

The *QWidget* class is an ancestor of all GUI elements in Qt. Thus, from now the *widget* will be a synonym for any element. *QWidget* forms an empty surface, a window with nothing inside like a clean sheet of paper. Qt offers many basic widgets like buttons, check boxes, or combo boxes, and I have not faced a situation in which I would miss any elementary widget. Usually, one needs a combination of a few elements where *QWidget* is suitable for holding the group of elements together. [23]

Within *QWidget* it is possible to place an element anywhere. In real cases, one usually needs some kind of alignment, e.g., all elements lined up in one row or column. Qt uses a system of layouts. Such layout may be attached to *QWidget* and control the alignment of elements inside the widget. There are four types of layout:

- *Box layout* - items are clustered in one row or a column.
- *Form layout* - there are two columns of items. It represents classic form where one row contains label in the first column and text input in the second one.
- *Group layout* - it arranges items in a grid.
- *Stacked layout* - it follows a strategy where only one widget is visible and the others are hidden. A programming interface for the item visibility switching is provided. [23, 24]

Once we have chosen the layout, we can add widgets to it.

5.2.2 Events

We understand widgets which enable us the construction of a static user interface. The word static means that nothing happens when you try to interact with the interface by mouse clicking. To bring a life to the elements, Qt uses a system of *signals* and *slots* serving as a tool for event handling, e.g., answering a button click. The *signal* is a medium for notifying about event whereas the *slot* reacts to these notifications. From the developer's point of view, the signal and the slot are nothing more than functions. Qt provides extra features beyond classic C++ - signal-slot interconnection and signal emitting. Once an event happens, the corresponding signal is emitted and caught by the interconnected slot. Imagine that we have a window with a button which should close the window. The button contains a signal that is emitted when users click the button. The window implements a slot function closing the window once the signal is emitted and caught by the slot. [25]

5.3 Connection of Qt and ESPRESO

It is time to finally start building our GUI. We will begin with an introduction of a class that will form a universal widget which can draw any *ECFObject* on the screen.

5.3.1 Base class

The class will be abstract providing the possibility of adjustment to specific *ECFObject* and related need for different GUI behaviour. Let us call the class *ECFObjectWidget*.

The main task of our class is to draw the contents of *ECFObject*, i.e., parameters. A parameter has assigned a data type which *ECFObjectWidget* uses to draw an appropriate widget. The drawing is done in separate methods for each data type. One can override these methods

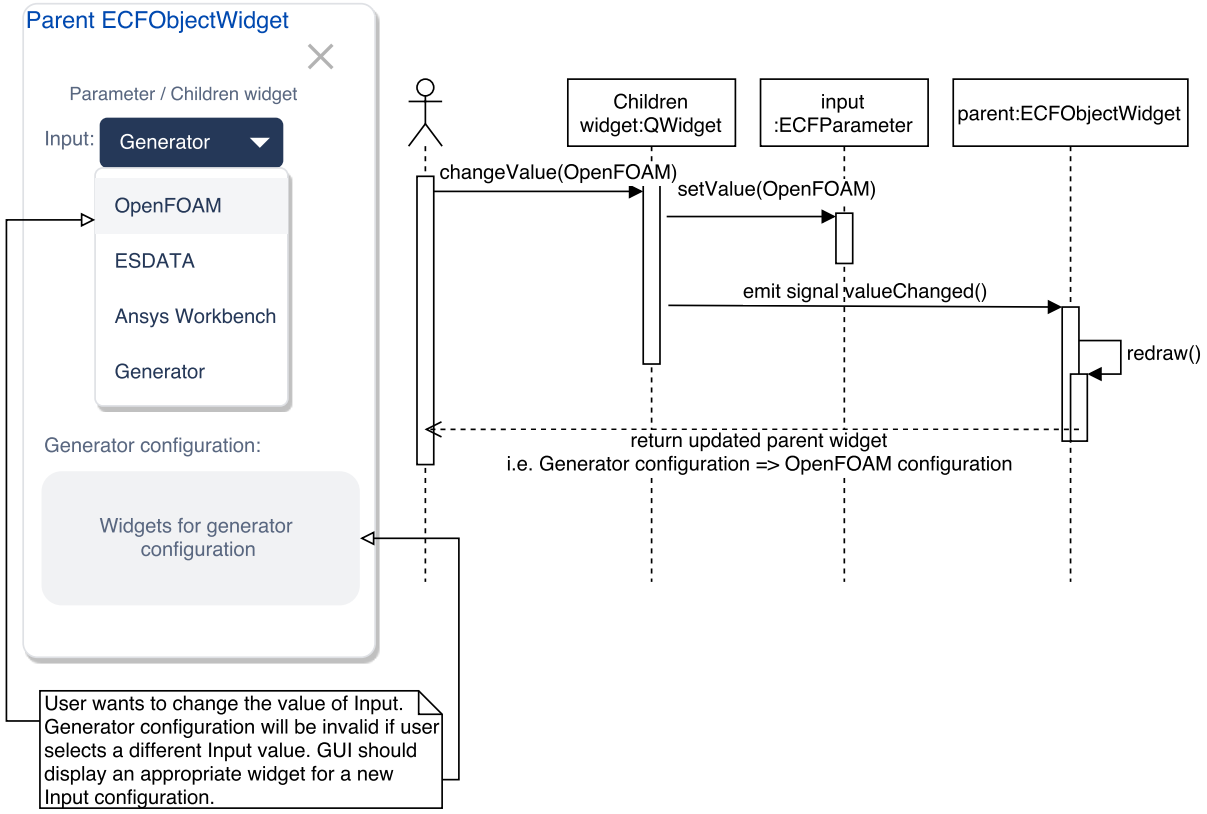

Figure 3: Sequential diagram with a GUI example showing a situation in which the change of the Input parameter requires redrawing of GUI

to customise the GUI behaviour for a specific *ECFObject*. Table 3 shows common data types in ESPRESO.

There exist some closely correlated parameters, the change of any of which influences the other one. This might require a change in GUI too, thus widgets that could cause such situation are connected with *ECFObjectWidget* via a slot. When a corresponding signal is emitted, *ECFObjectWidget* performs a new drawing of its parameters. This behaviour is described in Figure 3 with a simple example of a combo box, the value change of which influences the rest of GUI.

5.3.2 Validation and saving

ECFObjectWidget controls widgets for editing the parameters. Users should not be able to enter invalid data or at least save them. Therefore, there has to be a way how to prevent it. We will introduce an interface class *IValidatableObject*, which will provide methods for data validation including the check of content correctness and information about a found error. *ECFObjectWidget* will implement the interface, and, at the same time, it will also contain a list of such interfaces since it may contain another widgets requiring the validation.

Table 3: Selection of abstract data types used in ECF files

Data type	Description
BOOL	True/false value
STRING	Text string
INTEGER	Integer
POSITIVE_INTEGER	Positive integer <1, inf)
NONNEGATIVE_INTEGER	Integer <0, inf)
FLOAT	Real number
OPTION, ENUM_FLAGS	Set of values with only one selected
REGION	Name of existing region
MATERIAL	Name of existing material
EXPRESSION	Mathematical expression

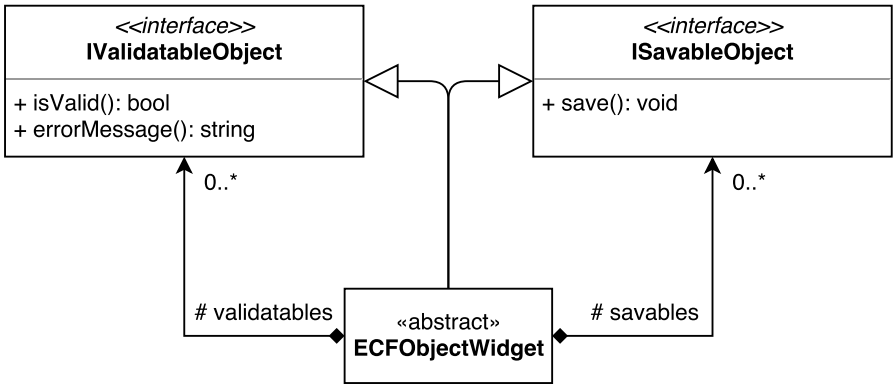

Figure 4: A class diagram of a basic widget wrapping *ECFObject*

If we have valid data in a widget, we can store them. For this purpose, we will present another interface *ISavableObject* enabling us to perform a save operation. In our context, it means that data from GUI elements are copied to the corresponding *ECFObject*.

Figure 4 contains a class diagram depicting the binding between *ECFObjectWidget*, *IValidatableObject* and *ISavableObject*.

5.4 Prototype of GUI

Before we get to particular descendants of *ECFObjectWidget*, it might be beneficial to clarify which widgets we will need to be present in GUI. We will not focus on small elementary widgets like text inputs, but we will concentrate on major logical blocks that are important for the basic work flow that we showed in Chapter 3.1.2.

If we scan the individual points in the use case, we may identify following blocks:

- 3D model.
- Materials.

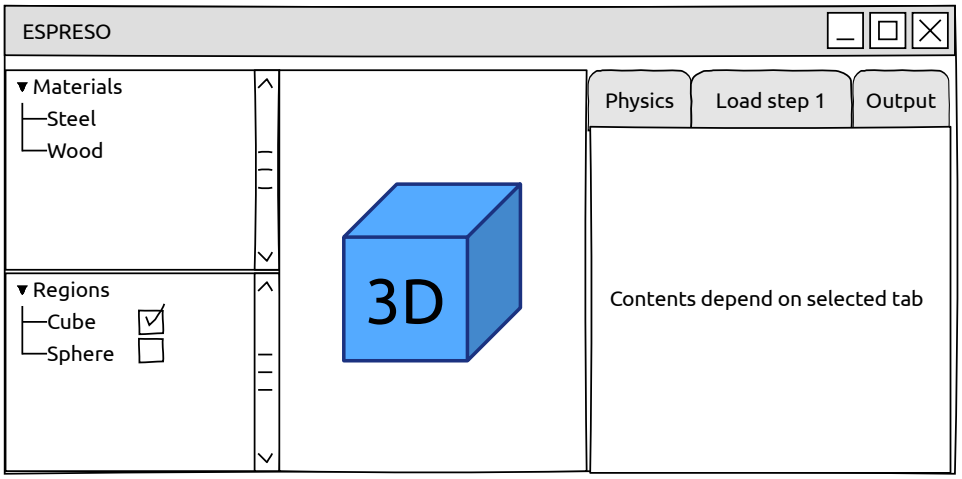

Figure 5: Early prototype (wireframe) of GUI

- Physics.
- Load steps.
- Output.

This set of blocks could be still reduced. First, physics, load steps, and output form one block since together they define a task that will be computed. Second, materials form an individual unit, even though their properties depend on physics. Nevertheless, we will see them as a library of materials that can be applied in various numerical tasks. Third, the 3D model will represent a block consisting of the model itself and the list of regions we have already mentioned in Chapter 4.2. A complete prototype of GUI is depicted in Figure 5.

5.4.1 Widget decomposition

On the previous lines, we have defined three main blocks - *3D model*, *materials*, and *work flow*. The last one unites physics, load steps, and output.

The *3D model* block takes care of the visualisation. Basically, we will need two widgets. The first one will control the 3D model rendering and the second one will contain a list of regions. From Chapter 4.2 we know that both widgets should be interconnected because of the region hiding and picking. With signals and slots, it is simple. When someone clicks on an object in 3D model, a signal is emitted and same happens when users change a state of any check box in the region widget. In terms of class dependencies, the signal-slot interconnection may be done within one of the widgets, thus only one widget will have to hold information about the other.

In this case it does not matter which one will maintain the interconnection. We will choose the region widget.

The *materials* should form some kind of a library independent of the current computation task. Nevertheless, this block will be more general and will represent a place where any reusable data can be defined. Hence, we may imagine that variables discussed in Chapter 4.1 will be located there too. The block name *materials* becomes misleading, therefore we will call it *data sets*.

The *work flow* forms a structure that takes control of a few other widgets. As we may see in Figure 5 it is a tabbed widget where every tab contains a part of ESPRESO configuration. These tabs will be the descendants of *ECFObjectWidget*. The figure with the prototype contains physics, load steps, and output. However, this is not the final list of tabs that will be present. We will need one more tab for an input mesh configuration where users will be able to choose the input format or directly configure the 3D model via the generator. This tab corresponds with the part of ECF that we discussed in Chapter 2.2.1.

3D model, *data sets*, and *work flow* are basic blocks/widgets which form our GUI. Nevertheless, we need something that would stick and hold them together. We will wrap them into another widget. Let us call it *work space*. It will not be just a wrapper that will take care of their positions only. *Work space* will resolve a communication between the individual blocks, i.e., reacting to signals. For example, if users change the input file in the mesh tab, a new 3D model should be drawn. *Work space* will be a mediator catching the mesh change signal and informing the *3D model*. Additionally, we will add two new buttons into the *work flow*, one for a loading of an existing ECF and the second for storing of current configuration to ECF.

Let us finish this section with two figures. First, Figure 6 shows a class diagram of all widgets that we have introduced in this section. Second, Figure 7 contains a prototype of GUI extended with the captions of blocks and new widgets.

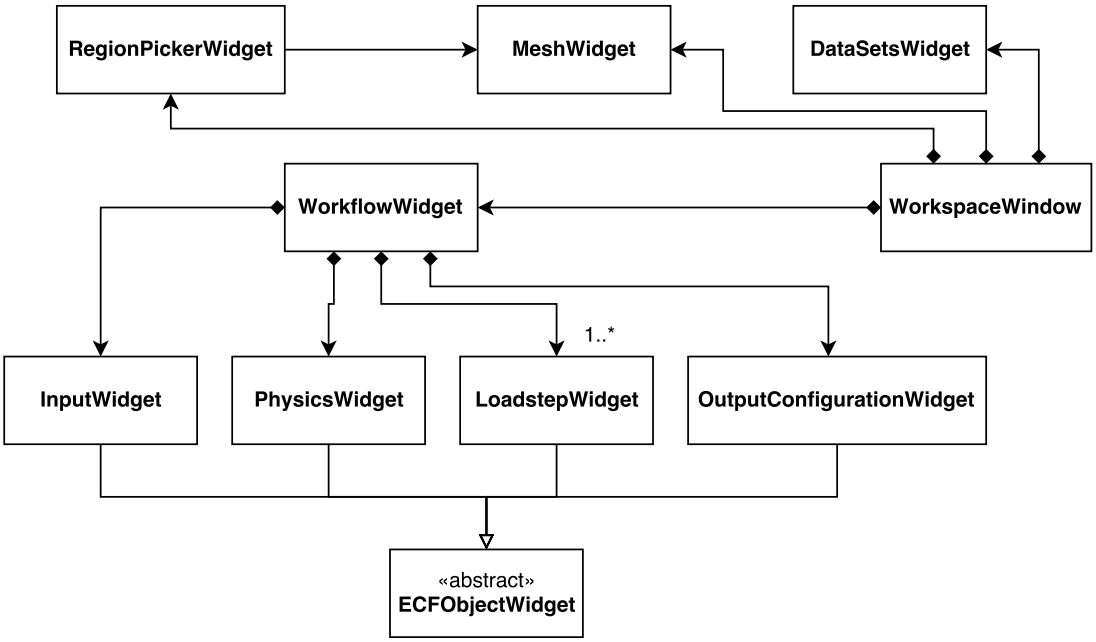

Figure 6: Class diagram of basic GUI blocks/widgets

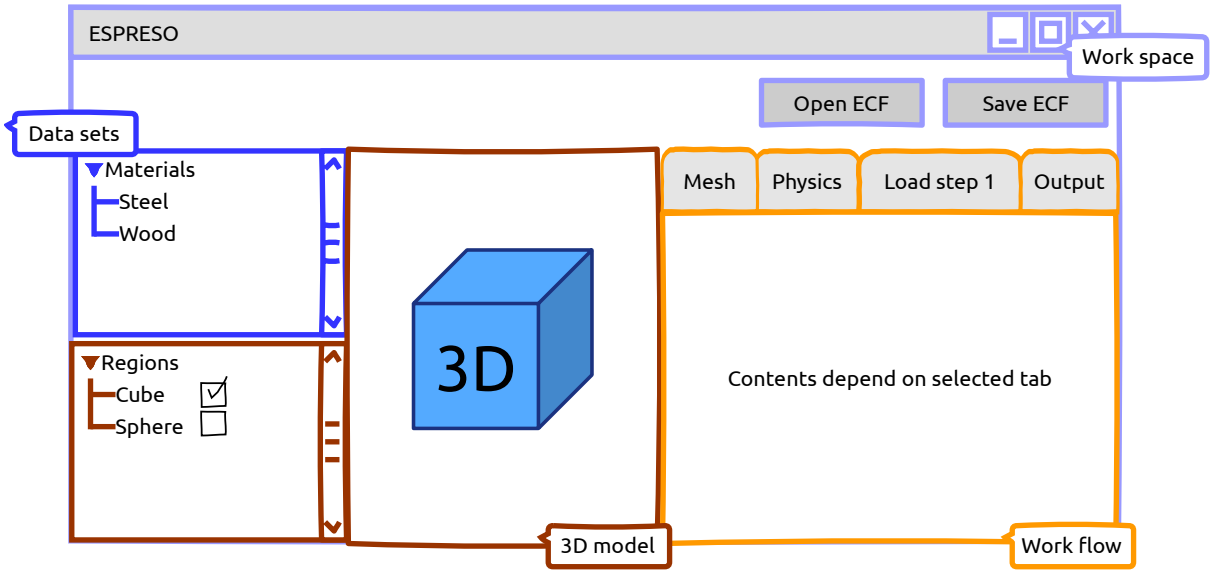

Figure 7: Extended prototype of GUI

6 Parallelization

In Chapter 4.3, we briefly specified the requirement of parallelization. In this section, we will explicate techniques which ESPRESO uses to run in parallel, and we will show concrete steps necessary for enabling the parallel execution of the solver directly in GUI.

6.1 Parallelism in ESPRESO

ESPRESO combines two parallel paradigms leading to a hybrid parallelism. The first paradigm is shared memory computing. A main process of application creates several smaller working units (i.e., threads) sharing one memory. This strategy uses the OpenMP (Open Multi-Processing) library providing the ability of the task division into threads. The main disadvantage of shared memory computing is the memory being accessible by any thread. Data inconsistencies might occur when two threads are trying to access same part of memory, and one of them wants to change the content. This situation is called race condition, and it is possible to solve it in several ways. However, we are not going to dive into it since it is not necessary for our purpose. [26]

Distributed memory computing is the second method of parallelization. In contrary to threading, application is divided into processes at the beginning of execution. Each process has own isolated memory. ESPRESO uses MPI (Message Passing Interface) library to enable distributed computation. MPI divides our program into specified number of processes which is immutable during the whole execution. Since the processes are isolated already from the application start we need some tool to exchange data between them. MPI provides functions for sending and receiving primitive data types like integers, bytes or doubles between processes. The distributed programming does not suffer from race conditions, on the other hand, data exchanges may be slow if communicating processes run on different computers and data has to go through their interconnection. [26]

Hybrid parallelism starts with the distribution into processes, then, every process could divide itself into threads. Figure 8 shows the distribution procedure.

6.2 Parallelism in GUI

GUI will only extend ESPRESO, meaning that the present code base of ESPRESO will be extended by GUI source codes. This close link between ESPRESO and GUI allows us to use the classes for the ECF manipulation, but, generally, we may access any class including the solver. In order to utilise all features of solver concerning the parallelism, it is not enough to just call the corresponding classes.

Support of threading does not require much work. Threads are created and destroyed dynamically during run time. We have to compile GUI with OpenMP support, i.e., add one extra flag (*-fopenmp*) to compiler, and before the execution environmental variables have to be set specifying how many threads should be used. [27, 28]

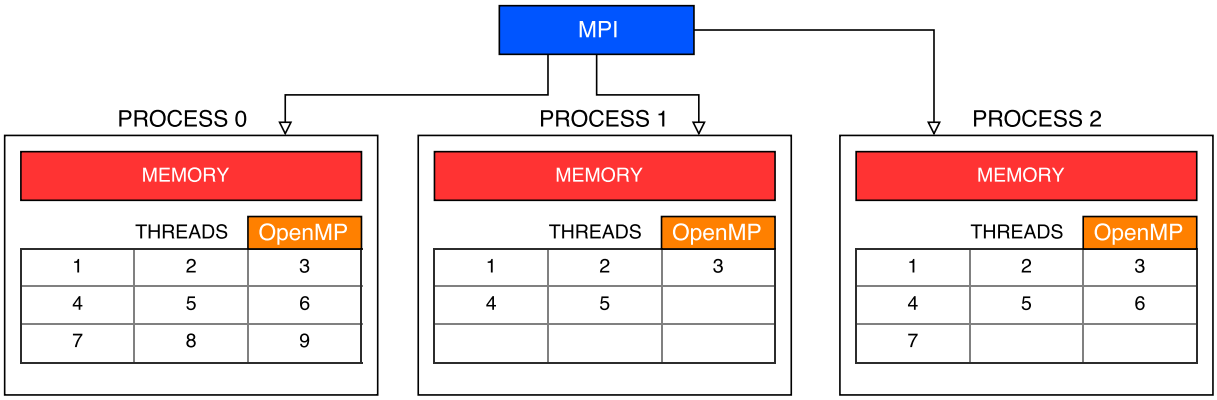

Figure 8: Hybrid parallelism with MPI and OpenMP

In the case of MPI, we will have to do more work. MPI application has to be compiled with a special compiler. Also, if we want to launch a program with more than one process, we have to use special command, which will distribute the program into more instances. [29]

If we launch GUI on n processes, there will be n opened windows. Obviously, this is not the behaviour that we want. We would like to run GUI in parallel, but it should be displayed on one process, and the rest of processes should be reserved for the computation. Thus, they will be idle until a computation is launched. There are more questions that we have to consider, therefore the following section is dedicated only for this topic.

6.3 Integration of MPI in GUI

We have come to the conclusion that one process will take care of GUI, and the rest will be computing. Nevertheless, it is not clear, whether users should launch GUI on n or $n+1$ processes, if they want to perform computation on n processes. In other words, we should decide if there should be a strict separation of GUI and computation processes, or if all processes would be computational, and one of them would also run GUI. Thus, when users start the solver, GUI would not respond to user events for the time of task. The confusion about the number of processes could be fixed by running GUI in a separated thread, therefore one process will be shared by GUI and the calculation. The final number of processes is n.

Let us move to the implementation. Processes are numbered from zero, therefore GUI will be always under the supervision of process zero. Thus, everything concerning GUI should be hidden for other processes. This is commonly achieved by wrapping the code that we want to hide into *if statement* which checks the number of current process. [29]

We will need a tool for controlling processes and exchanging messages between them. GUI lies in a master process which controls everything. It should wake up the computational processes and give them a command to start computation. Therefore, we will introduce a new *MpiManager* class providing methods for the process controlling.

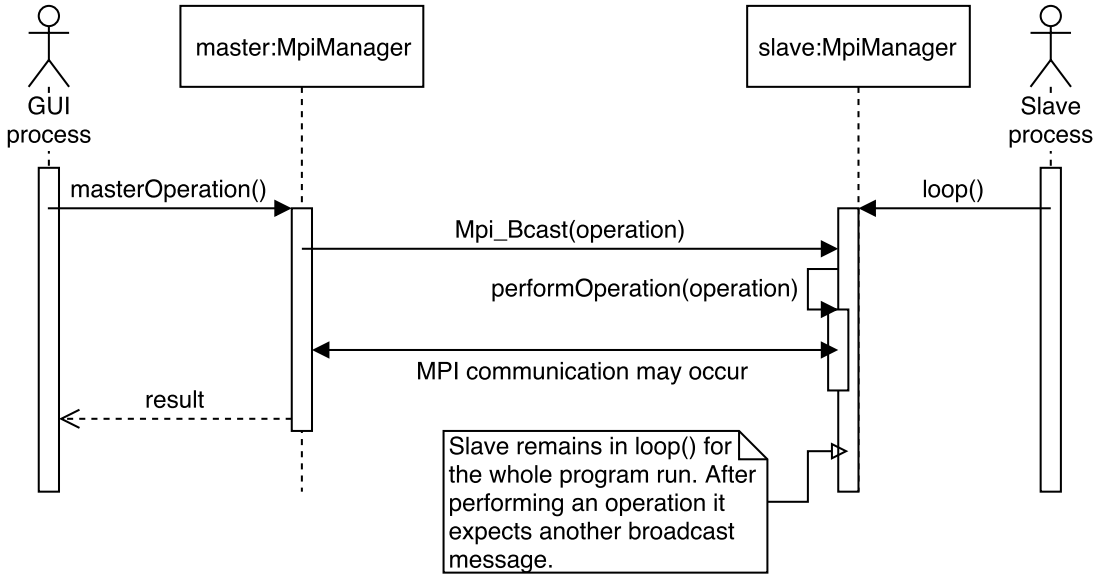

Figure 9: Cooperation of master GUI process and computational workers

6.3.1 MpiManager class

Every process has own instance of *MpiManager*. The master process (i.e., GUI) uses *MpiManager*'s methods for controlling and sending messages to the slaves (i.e., computational processes). A slave's scenario is simple - waiting for command, executing command and waiting again. Thus, this strategy forms the *loop*. The command is distributed via a broadcast message (*MPI_Bcast*) to every slave. On the basis of received command, a corresponding operation is performed by slaves. These operations are implemented by *MpiManager*, which also provides the *loop* method. Commands are represented by master methods in *MpiManager*, i.e., methods which may be called only by GUI.

A complete scheme of the cooperation between master and slaves is depicted in Figure 9. The method *masterOperation()* represents any possible operation.

6.3.2 Distributed operations

Now, we will show the work we are going to give out to slaves. Following list specifies basic operations:

- Each computational process requires complete information about the configuration. Thus, *ECFRoot* which users edit via GUI has to be copied to slaves.
- After every slave obtained the complete configuration data, the solver could be launched.
- ESPRESO's mesh generator defines which part of mesh should be present on a particular process. Hence, when GUI loads an ECF with a mesh produced by the generator, it should

obtain mesh parts from each process and build a complete 3D model that can be visualised. More about mesh rendering in Chapter 7.

7 Visualisation of 3D model

We have became familiar with many libraries and technologies including ESPRESO, Qt, MPI, and many more. This chapter provides a brief overview of OpenGL (Open Graphics Library), its usage within Qt, and application in GUI together with ESPRESO.

7.1 OpenGL

OpenGL is an API specification for manipulation with graphics and images. It specifies a large set of functions and defines exact output of each function. Usually, developers from graphics card manufacturer companies implement these functions. [30]

Today's modern OpenGL is based on version 3.3, which was released in 2010 and brought a different programming model from earlier versions. All new versions after the release of 3.3 only extend API and do not change it. Hence, GUI will not support lower versions of OpenGL than 3.3. [30]

Following sections give a brief overview of OpenGL principles to an inexperienced reader. On the other hand, the overview does not explain everything necessary to perform rendering in OpenGL. If you need more information, there is a very good guide in [31].

7.1.1 State machine

OpenGL behaves as a large state machine. The programming starts with the setting of various options and flags, which define what will be rendered, and how the drawing will be performed. After the configuration, programmers can call the functions designated for the rendering. [30]

7.1.2 Primitives

Vertex is a basic graphical element that is used in OpenGL, i.e., a vector of coordinates x, y, and z which define a point in 3D space. When we have a set of vertices, we may choose if OpenGL should draw points, lines or triangles. In the case of lines, vertices are processed in pairs, whereas, if we wanted triangles, the vertices would be processed in triplets. Points, lines and triangles[1] form a basic set of shapes that OpenGL can draw. They are called primitives. If we needed to draw more complex shape, we would have to build it with these primitives. The triangles are the most used to construct larger objects. There are several reasons why a triangle is the best option, and you may find it for example in [33] (p. 108, section Triangles). [32]

[1] OpenGL supports also various combinations of same primitive, e.g., a loop constructed from lines, a stripe of triangles or lines, etc. Older OpenGL APIs before version 3.1 enabled to draw squares (quads), but this functionality has been removed from newer versions. [47]

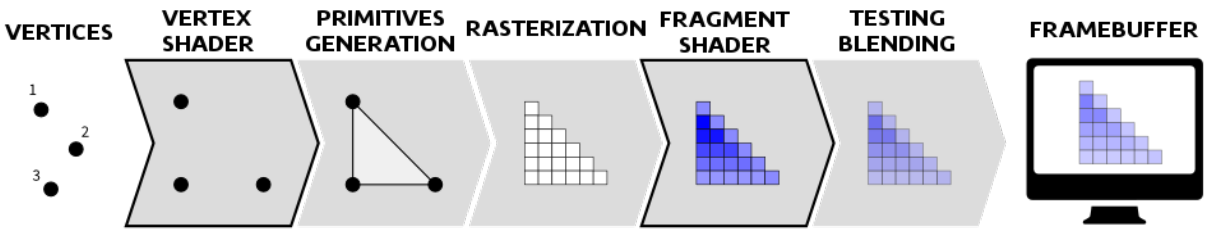

Figure 10: OpenGL pipeline [46]

7.1.3 Shaders and programs

Every vertex sent to a graphics card is processed by a set of shaders. *Shader* is a program that is executed on the card. It is programmed by a special language, OpenGL Shading Langugage (GLSL), which uses similar syntax to C++. Different shader types are ordered in *pipeline* (see Figure 10). Every shader in the pipeline produces an output that is used as an input for the subsequent shader. Every vertex goes through every single shader in the pipeline, before it is finally drawn on the screen. Each shader type has a specific role, e.g., *fragment shader* takes care of the final colouring of pixels, before they are rendered. We may use default shaders or replace them with own shaders described with GLSL. In order to make the replacement, we have to construct *shader program* which represents the pipeline. Then, we can assign our shaders to the program and tell the card to use it instead of the default shaders. [32]

So far, we have presented fragment shader. It controls the pixel colouring. Also, it is the right place for computing special effects, i.e. shading. The pipeline consists of more shaders but they are not important for our needs. [32]

7.2 OpenGL in Qt

Qt provides tools for working with OpenGL. The classic OpenGL API does not follow object oriented paradigm. It is a set of functions, thus it looks more like a library for C language than for C++. Nevertheless, Qt introduces a set of classes which tries to bring object oriented principles. It does not substitute the classic API completely, rather it simplifies work in some situations. [34]

Typical example is the creation of shader program. In classic API, one has to create it via a special function, which returns an identification number of the new program. When programmers want to manipulate with it, they have to call an appropriate function and pass the identification as an argument. In the case of Qt, one can create an instance of shader program class and call methods for the manipulation. This approach frees programmers from calling concrete OpenGL functions, instead of them more abstract methods are provided hiding the usage of OpenGL. Qt authors offer this abstraction to ease cross-platform development for mobile and embedded devices. For desktop application development, it brings at least a benefit in the form of object oriented approach. [32, 35]

Listing 3 shows a manipulation with shader program and fragment shader via both, the classic OpenGL API and the Qt object oriented alternative.

```
// COMMON PART FOR BOTH APPROACHES
const char * fragmentShaderSource = "... shader source code in GLSL ...";

// Classic OpenGL API
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);

unsigned int shaderProgram;
shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);

// Qt
QOpenGLShaderProgram shaderProgram;
shaderProgram.addShaderFromSourceCode(QOpenGLShader::Fragment, &
    fragmentShaderSource);
shaderProgram.link();
```

Listing 3: Comparison of classic OpenGL API and Qt alternative applied on shaders and shader programs [32, 35]

7.2.1 OpenGL and widgets

We have already become friendly with the cornerstone of Qt GUI, i.e., widgets. Qt features a special *QOpenGLWidget* widget class for handling manipulations with OpenGL. It introduces three methods *initializeGL()*, *paintGL()*, and *resizeGL()*, that can be overriden. Only within these methods, one is allowed to perform any operation with OpenGL. First, *initializeGL()* is executed only once at the beginning of *QOpenGLWidget* initialization, thus it is suitable for the 3D scene preparation purposes. Second, *paintGL()* is called periodically to render 3D objects when necessary, therefore it should be used for regular drawing of 3D objects. Third, *resizeGL()* should carry out any requisite task when the widget is resized. [36]

7.3 Implementation

In Chapter 5.4.1, we presented the class *MeshWidget* without any specific details. This class is a descendant of *QOpenGLWidget* resolving 3D model rendering via OpenGL.

7.3.1 Decomposing model into triangles

Before we can start drawing 3D objects on the screen, we must have prepared model data in an appropriate form, i.e., a 3D object defined by a set of vertex triplets representing triangles. ESPRESO implements functions for the decomposition of a 3D model into triangles. We will use them.

We know that a model may be divided into regions. ESPRESO provides tools to distinguish where each triangle belongs. Thus, we will maintain a dictionary of regions, which will contain information about a region name and a list of its triangles. Since in Chapter 6.3.2 we clarified that in the case of parallel execution the mesh could be divided into same number of parts as running processes, the whole conversion of model into triangles cannot be done only in *MeshWidget* in GUI process. We will have to perform the conversion in every process and gather the results in the master (GUI) process, which will render it.

7.3.2 Regions

In the previous section, we have introduced the dictionary of regions. The dictionary can be used in combination with *RegionPickerWidget* (Figure 6) to determine which region should be visible on the screen, and which one should be hidden. This is an expected feature that we declared in Chapter 4.2. Moreover, every region has to have a different colour, therefore we add this information to the dictionary.

7.3.3 Model manipulation

All software, which features 3D model visualisation, usually enables model rotation and zooming. Our GUI is not be an exception. Since this manipulation is something that was programmed many times, we adopt an approach from [37] (sections Mouse input and Zoom).

7.3.4 Region picking

Region picking cannot be handled completely by the signal-slot system, because only *QWidgets* and their derivatives may use it. Although, *QOpenGLWidget* inherits from *QWidget*, it is only capable of detecting a mouse event (e.g., a mouse click) with the information about a precise position (i.e., coordinates x and y), where the event occurred, but there is no built-in tool for the recognition of concrete 3D object hit by mouse. 3D objects are not *QWidgets*. They represent only a set of coloured pixels drawn in *QOpenGLWidget*. Thus, we have to provide own strategy to detect whether users clicked on a region. [36]

This issue can be solved by an algorithm based on *colour coding* described in [38]. This method colours every pickable object with a different colour. Qt provides information about a click position, and OpenGL features a function for obtaining the colour of a pixel in specific position. If we painted every region with different colour, we would solve the region picking problem.

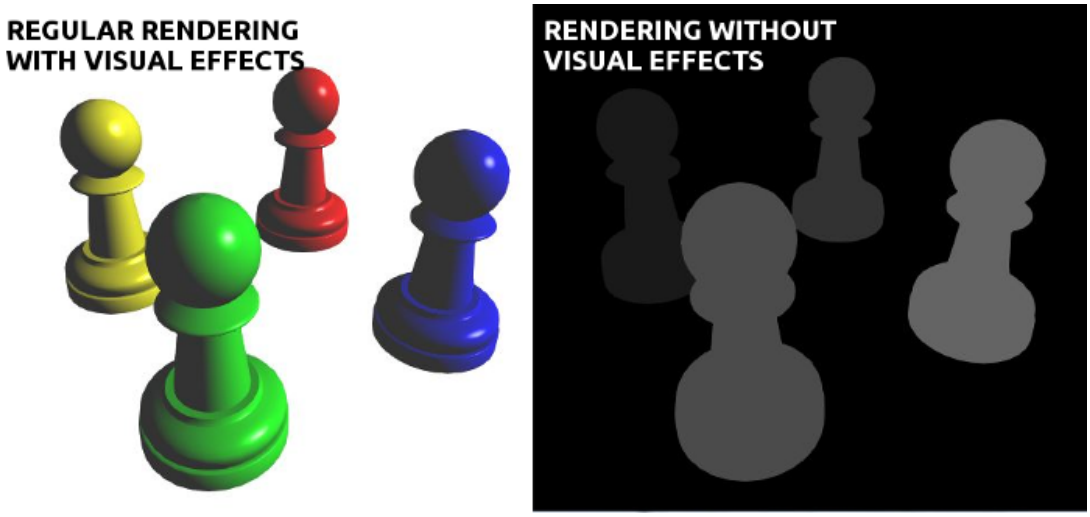

Figure 11: Regular rendering featuring visual effects and basic rendering [38]

The proposed method suffers from the possible lack of colours, if the number of regions would be extremely high. OpenGL uses classic the RGB (Red Green Blue) colour model, where each colour component is represented by a number from 0 to 255, thus total number[2] of all possible colours is 16581375, i.e., 256 possibilities per each component = 256^3. If we decrease this number by one, we get a limit of region count, because higher number of region would result in a colour collision. We decreases 16581375 to 16581374, since one colour is reserved for the background colour. ESPRESO development team agreed with this limitation. Computational tasks, which they currently solve, contain tens of regions.

One may notice that we have already spoken about colouring each region distinctly in Chapters 4.2 and 7.3.2. Thus, it might seem that we have already stood halfway to the solution. However, we have not. The colouring for the region picking will be performed separately, i.e., hidden for users' eyes. The reason is that the regular painting features variety of visual effects such as a light to provide better look. Unfortunately, the effects imply a possible change in region colour. In other words, the colour of a region might vary in different parts of the region due to the effects. Therefore, when users click on 3D model, GUI draws the regions with a fragment shader freed from the effects, then, it obtains a colour of pixel which users selected, and compares it with a list of region colours. If there is a match, *MeshWidget* emits a signal informing about the picked region. The signal is caught by *RegionPickerWidget*.

The difference between the regular rendering with visual effects and the basic drawing is depicted in Figure 11. Each chessman might be understood as a region. We may see that the surface of chessmen has variety of colours due to the light effect. In contrary, the right part of the figure features chessmen, each with one solid colour.

[2]The reference article [38] shows a procedure with only one colour component. We extend it by all three components, i.e., red, green, and blue.

To sum up, for the region picking we need to introduce own procedure that combines features of Qt and OpenGL. We implement a method based on the colour coding. Because of the visual effects included in the regular painting of 3D model, we have to switch to new fragment shader specialised for the picking, that does not apply the effects which influence colours.

8 Implementation of GUI

Source codes of GUI are available on the attached CD, the content of which is described in Appendix A. This chapters shows more details concerning the implementation of the final solution.

8.1 Widgets for expressions

In Chapter 4.1, we specified three structures, i.e., the expression, the table, and the piece-wise function.

Rules for defining expressions proceed from a notation used by the *ExprTK* library [39], which ESPRESO applies internally for the validation and the evaluation of expressions. The notation is very simple. For example, we could show a simple quadratic function: `x^2 + x + 1`.

Generally, the expression is a relatively short string, thus we can represent it by a text input (*QLineEdit* [40] class in Qt) and check its correctness via *ExprTK*, before it is saved into the corresponding *ECFValue*.

8.1.1 Table

We determined that the table is a set of couples $(x, f(x))$. Both x and $f(x)$ are specific values, i.e., real numbers defining the function. Graphically, each of them is a column in the table. For displaying data in tables, Qt offers the *QTableView* widget. In order to restrict table contents only to real numbers, one may use delegates. The *delegate* (*QItemDelegate* [41]) is a structure which is called whenever users try to edit a table cell. Delegate is responsible for creating a widget for cell editing. The default widget is the *QLineEdit* text input. This class enables to restrict its contents via regular expressions. Thus, we may create own delegate offering a text input guarded by a regular expression forcing users to type real numbers only. Qt uses the Perl notation [42] for regular expressions. Our expression for real numbers would look as following: `^[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?\$`.

Let us return to ECF files. ESPRESO uses a simple notation for tables. Every table starts with *TABULAR* keyword followed by the list of $(x, f(x))$ couples. If we transferred Table 1 into a raw ECF form, we would obtain: `TABULAR [ 0,10; 1,20; 2,30.6 ]`.

All this behaviour is hidden in the *TableTypeWidget* class.

8.1.2 Piece-wise function

The piece-wise function does not differ from the table too much. From the graphical point of view, it is just another table with more columns. If we look in Table 2, we may observe these important components:

1. Left parentheses/angle brackets - left-open/left-closed
2. Value of left bound

3. Value of right bound

4. Right parentheses/angle brackets - right-open/right-closed

5. Expression/function

The individual points above represent columns. Basically, cells of all columns are only text inputs. Nevertheless, in order to provide better user comfort and validation, a special delegate for each column was developed. For columns 2 and 3, delegates are same as in the case of the table from the previous section. For columns 1 and 4, we need a new delegate which displays a combo box with two options, i.e., "(", "<" or ")", ">". The last column should provide a basic text input for the expression. It is validated by *ExprTK* when users attempt to save the whole piece-wise function. Thus, the last column does not require any delegate.

The piece-wise function is not currently supported by ECF format. Nevertheless, apart from the expressions, *ExprTK* provides control structures like *if-statement* known from programming languages. We can easily represent the piece-wise function with a combination of several if-statements. Listing 4 shows an example of transferring Table 2 into the *ExprTK* format. This way of implementing the piece-wise function does not require any changes in the ECF format and ESPRESO, since the function will behave like the expression. Only GUI has to distinguish the piece-wise function from the expression, but this is not difficult. If we consider that the function contains keyword *if*, whereas the expression does not.

The piece-wise function is implemented by the *PiecewiseTypeWidget* class.

```
// FIRST ROW
if (x > -inf and x <= 1)
{
    x^2;
}
// SECOND ROW
if (x > 1 and x < 2.5)
{
    x + 1;
}
// THIRD ROW
if (x >= 2.5 and x < inf)
{
    sin(x);
}
```

Listing 4: Transferred Table 2 into ExprTK if-statements

8.1.3 Generalisation

Since the table and the piece-wise function are graphically just special cases of a table with different number of columns and data type inside them, we may construct a more general class (widget), which would wrap the *QTableView* class and take control of common behaviour of the table and the piece-wise function, i.e., mainly the manipulation with the data inside the table widget. This widget has name *TableWidget*.

8.1.4 Delegates

In closing, we return to delegates. In the case of the table and the piece-wise function, we have used them to validate cells with real numbers. Because Qt use them very often, I have considered beneficial to construct a delegate with text input that provides an universal interface for the specification of allowed input data. In Chapter 8.1.1, we used the regular expression for that. Regular expressions are represented by the *QRegExpValidator* class in Qt, and this class inherits from the *QValidator* class [43]. The text input class *QLineEdit* enables us to assign it *QValidator*, which features functions for the determination of forbidden input data. Our universal delegate accepts any *QValidator*'s descendant. It is important to take into consideration that every time users attempt to modify a cell, a new *QLineEdit* object together with *QValidator* are created. Therefore, we have to deliver an interface offering the creation of appropriate *QValidator* to the delegate. This could be smartly solved by factory design pattern. Resulting class diagram and communication scheme of the new classes are depicted in Figures 12 and 13.

8.2 Overview of widgets

In the previous chapters, we have introduced several widgets which together create complete GUI. In this section, we are going to summarise them and present a few new elements that have been implemented during the implementation phase of the GUI development. The following subchapters try to provide a comprehensive view for any interested reader, who would like to study the source codes of GUI. On the other hand, they do not form a fully-fledged documentation of code.

8.2.1 Main blocks

This section gives an overview of widgets that we got acquainted with in Chapter 5.4.1.

- *WorkspaceWindow* - it represents the main window which connects all subsequent widgets together.
- *WorkflowWidget* - it is a tabbed widget that unites widgets for the configuration of input (*InputWidget*), physics (*PhysicsWidget*), load steps (*LoadstepWidget*), and output (*OutputConfigurationWidget*).

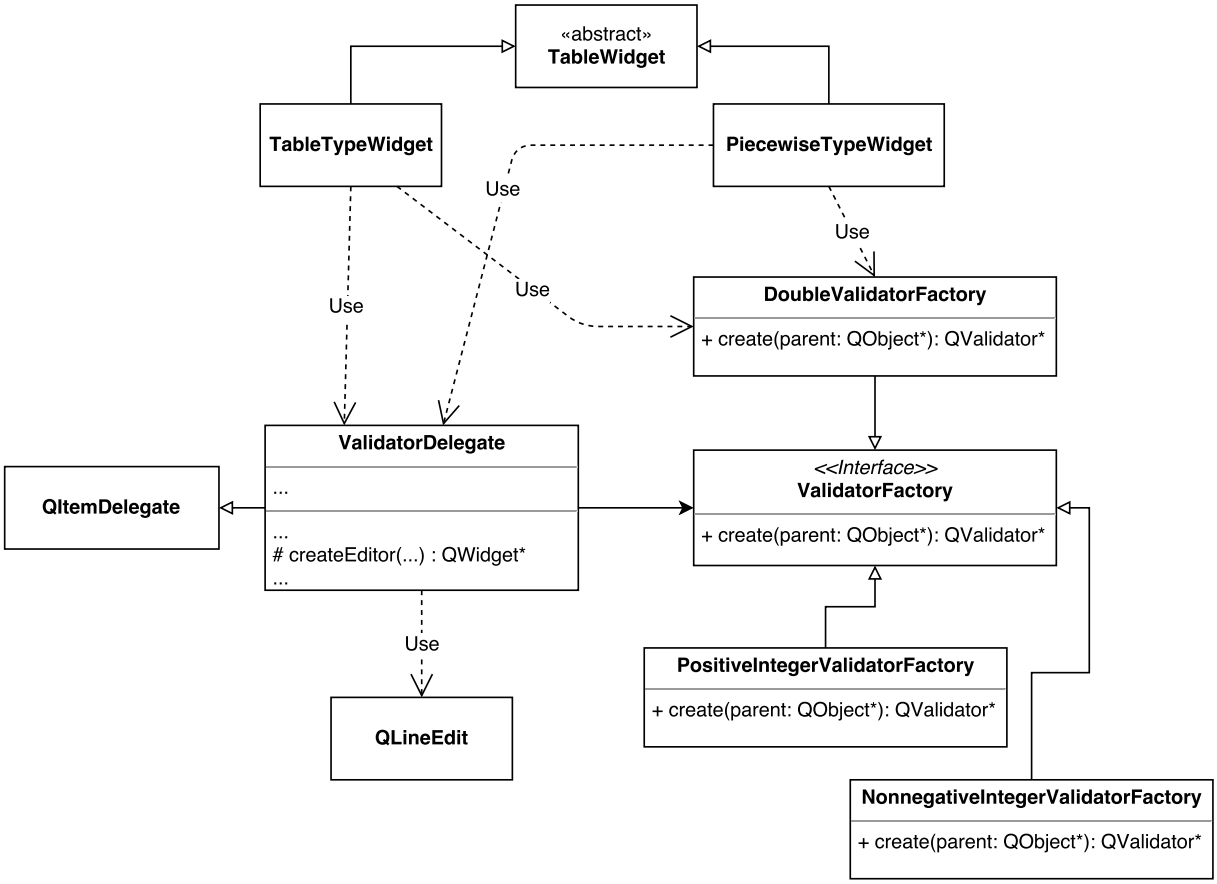

Figure 12: Class diagram of the text input delegate with the universal interface for setting a validator based on factory design pattern

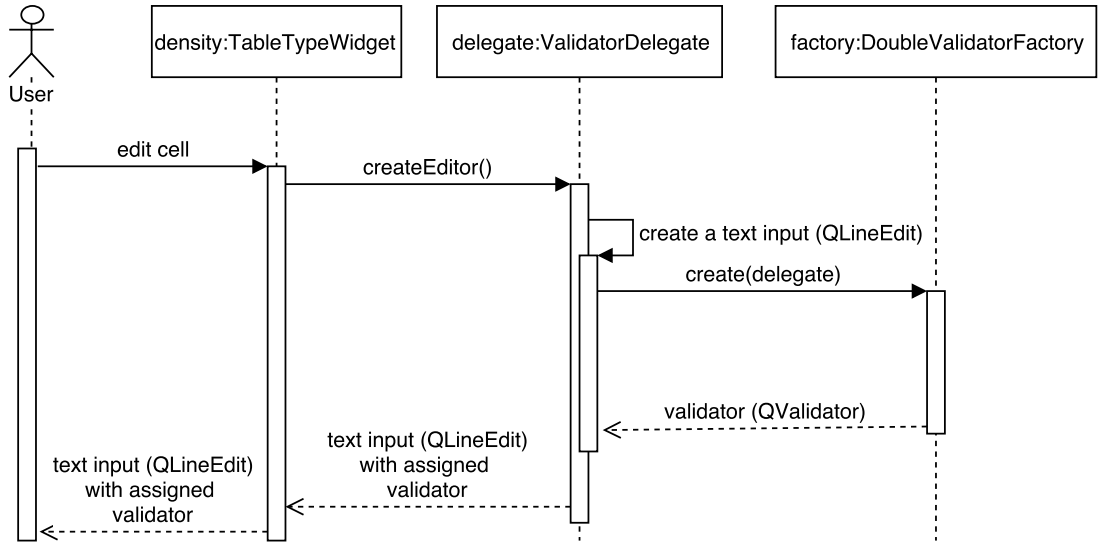

Figure 13: Sequential diagram depicting the creation of a text input for the table cell editing with delegate and validator

- *DataSetsWidget* - this widget provides a tree-like structure of its content, and it is intended for the definition for any reusable data during the configuration of ECF. Currently, one is able to define materials there.

- *MeshWidget* - a place where all 3D rendering of input mesh/model is performed. It accepts a signal from *RegionPickerWidget* notifying which region should be hidden.

- *RegionPickerWidget* - it contains a list of all regions. Users are able to select regions that will be rendered by *MeshWidget*. It accepts a signal from *MeshWidget* notifying which region users clicked on and informs users about that.

8.2.2 Abstract structures

In Chapter 5.3.1, we devised the *ECFObjectWidget* class, which represents an ancestor of all widgets that should take care of a specific *ECFObject*. The following classes and widgets are designed to form starting point for a whole range of complex structures.

- *IValidatableObject* - this interface unifies structures, the contents of which can be validated, e.g., a form for configuring physics implements this interface to provide a possibility to check the correctness of input data.

- *ISavableObject* - this interface constitutes a basis for any object requiring methods for saving its state. Specifically, in our case, it creates a tool for moving data from GUI elements into *ECFObjects* and *ECFValues*.

- *ECFObjectWidget* - an abstract class which enables rendering of *ECFObject*.

- *FixedECFObjectWidget* - it inherits from *ECFObjectWidget*, and it is the simplest widget that can be directly used to draw an *ECFObject*. It is suitable for applications in all areas where enough space for all elements inside *FixedECFObjectWidget* is guaranteed. If it is not, some elements might stay hidden to user's eyes.

- *ScrollECFObjectWidget* - it is an alternative to *FixedECFObjectWidget* which solves the issue of a want of space by the implementation of a scroll bar. Thus, users may scroll to reach hidden elements.

- *ECFObjectTreeWidget* - some *ECFObjects* behave as a container for other *ECFObjects*. Materials are a typical example. They consist of one object serving as the container which contains particular materials. Imagine, that we have several different containers, and we want to display them together inside one GUI element but their contents should be separated. A tree seems to be the most suitable widget for this case in Qt. Each container would be a direct child of the tree root, and the objects located inside the container would be children of the container node. In practise, it looks like the *data sets* block in Figure 7,

where *materials* represent the container node, and *steel* and *wood* are the children nodes. *ECFObjectTreeWidget* wraps the Qt tree widget with general functions for working with *ECFObjects* including possibilities of adding a new one, editing existing ones, and deleting them. This class is abstract and *DataSetsWidget* implements it.

8.2.3 Widgets for blocks of ECFValues

The configuration of ESPRESO includes many parameters which are represented by *ECFValues* in the simplest form. Usually, these parameters are located in an *ECFObject* in a number greater than one. For such groups of *ECFValues*, I implemented the *ECFParameterTreeWidget* class that arranges the parameters into a tree. Every tree node features a GUI element appropriate to the parameter's data type (see Table 3), e.g., in the case of OPTION, it displays a combo box with an expandable list of options.

8.2.4 Widgets for region properties and boundary conditions

There exists a special group of *ECFObjects* which behave as a dictionary, i.e., a structure with key-value pairs. Such *ECFObject* features two data types or two descriptions inside its metadata. In C++, the dictionary is implemented by the `std::map` class.

The main difference between a classic *ECFObject* and the dictionary *ECFObject* resides in a homogeneity of their contents. The dictionary guarantees that all its contents are of same parameter type (i.e., *ECFValue* or *ECFObject*) and data type, whereas the classic object may contain a mixture of values and objects with various data types.

Region properties and boundary conditions are typical examples of dictionaries. In both cases, the key-value pair is the property/condition, and the key is always a name of region. The value is the expression in the case of *ECFValue*, but if a property is a dictionary of *ECFObjects*, the value is represented by all parameters of that object. There is one dictionary for every single property/condition, i.e., one property is a key-value pair of a region (key) and the specification (value) of the property. GUI features following classes for the drawing of region properties and boundary conditions:

- *RegionPropertyWidget* - a descendant of *ECFObjectTreeWidget*. It arranges the properties into a tree. The children of root are the properties, and their children are the regions with assigned definition of the property.
- *RegionPairDialog* - it is a dialog window that appears on the screen whenever a users want to add a new property or edit existing one.

8.3 Compilers

ESPRESO officially supports the compilation with two distinct compilers, i.e., GCC and Intel. GCC has to be combined with a MPI compiler such as OpenMPI or MPICH. On the other hand, Intel offers a complete suite with compilers for both C/C++ and MPI.

During the implementation of GUI, I have encountered one issue regarding Qt compiled by the Intel compiler. We have already mentioned the signals and slots that are used for event handling in Qt. If we want to link signal and slot, we have to apply a `connect` method. There exist two syntaxes of the method, a new one (*functor-based*) introduced with Qt 5, and an old one (*string-based*). Listing 5 shows both notations. The old one is still fully supported, thus developers may choose which one they would use. Nevertheless, in the case of Qt compiled by the Intel compiler, the new syntax does not work, meaning that it is possible to perform the compilation, however, the connection of signals and slots fail during runtime. The reason, why this happens, is described in [44]. In short, it is a problem of linker and different memory arrangement in comparison to GCC. According to Qt's bug tracker [45], it seems that this issue has been fixed by version 5.9.0, and if someone use an older version they have to rebuild it with a special flag turned on. Nonetheless, I together with EDT have decided to use the old notation, since it does not suffer from the problem, and during the development, I have not come across any limitations. Moreover, this solution does not place any demands on users, who would have to rebuild Qt in the case of the use of the new syntax.

```
// We create a button that changes contents of a text input
// when someone clicks on it. We have an imaginary widget
// with a function changeTextInputContent() which performs the change.
QPushButton* button = new QPushButton;
MyWidget* widget = new MyWidget;
// OLD SYNTAX
connect(button, SIGNAL(pressed()), widget, SLOT(changeTextInputContent()));
// NEW SYNTAX
connect(button, &QPushButton::pressed, widget, &MyWidget::
    changeTextInputContent);
```

Listing 5: Example of the new notation and the old one for the connect method in Qt

8.4 Omitted features

In Chapters 3.1.1 and 4, we determined various requirements that GUI should meet. This section summarises features, which have not been implemented yet.

8.4.1 Variables

In Chapter 4.1, we discussed expressions together with the variables, which behave as a reusable container for the expression data types that may be applied everywhere, where an expression is expected, just like a variable in classic programming languages.

Variables have not been implemented. Nevertheless, it is expected that users would be able to define them in the *data sets* block. The ECF format supports the variables, however, they can hold any string. In other words, their content is not limited to expressions only.

8.4.2 Parallelism

The current version of GUI implements the *MpiManager* class that we introduced in Chapter 6.3.1. However, I have integrated only the last point concerning the mesh distribution from the list in Chapter 6.3.2. Also, the sharing of zero process between GUI and a calculation has not been included yet. The sharing will be necessary for the remaining points in the list of distributed operations.

9 Example of heat transfer computation

This chapter demonstrates the main output of this thesis, i.e., GUI for the ESPRESO configuration. GUI capabilities are demonstrated on a basic computation example of heat transfer.

9.1 Cooler example

This example shows how to compute linear temperature distribution in an aluminium cooler with GUI. The cooler is shown in Figure 14. The cooler's body is divided into two regions, i.e., *BOTTOM_NODE_SET* and *FACE_SET_01*[3]. The additional *PART_01* region, which is not depicted in the figure, represents the whole cooler.

BOTTOM_NODE_SET has constant temperature 80°C, whereas *FACE_SET_01* starts with temperature 22°C. The goal is to compute heat transfer from *BOTTOM_NODE_SET* to the rest of cooler's body. The computation requires additional physical properties to be specified. They are summarised in the list below. Their meaning is not important in the context of this thesis. The main aim is to show how one can set these properties via classic ECF and via GUI.

- 3D linear steady state problem
- Material configuration:
  - Aluminium alloy
  - Isotropic model
  - Thermal conductivity $\lambda = 154\ [W * m^{-1} * K^{-1}]$
  - Set to the *PART_01* region
- Uniform initial temperature 293.15 $[K]$
- Constant temperature 80 $[°C]$, set to the *BOTTOM_NODE_SET* region
- Convection
  - Set to the *FACE_SET_01* region
  - Ambient temperature $T_{amb} = 22\ [°C]$
  - Heat transfer coefficient $HTC = 80\ [W * m^{-2} * K^{-1}]$

[3]ESPRESO uses two distinct groups of regions, i.e., element regions and boundary regions. Boundary regions can divided into subgroups, i.e., face regions, edge regions, and node regions. For example, *BOTTOM_NODE_SET* is a node region, *FACE_SET_01* is a face region, and *PART_01* is an element region. The meaning of these terms is out of the scope of this thesis. However, it is important to mention that the face regions and the element regions can be directly drawn with the OpenGL triangles which GUI is prepared for. The rest is not currently supported by GUI.

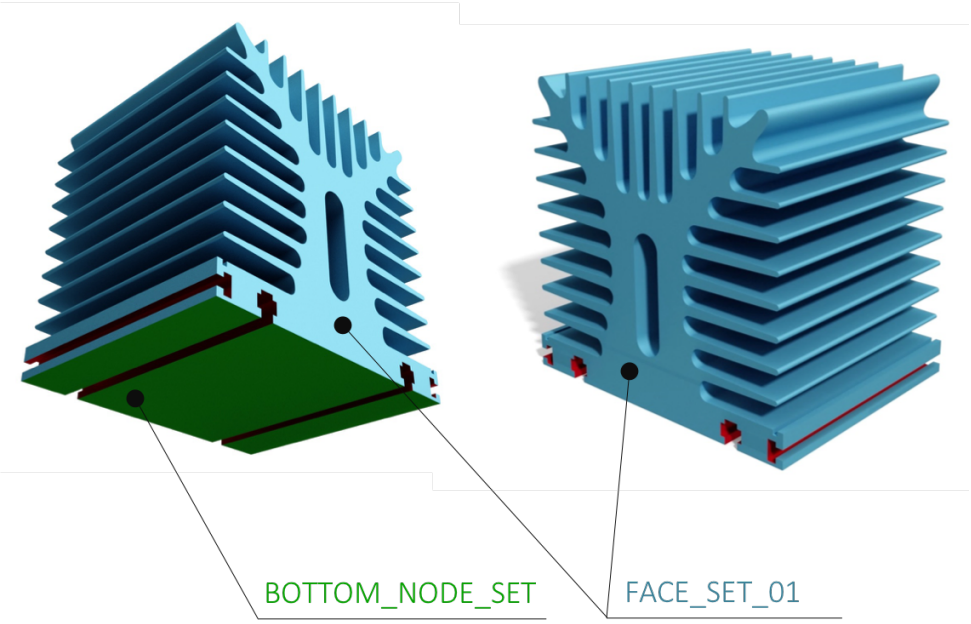

Figure 14: Aluminium cooler with region labels

9.1.1 Configuration via ECF

Listing 6 shows the ECF file for the Cooler example. It contains several parameters that we have not met yet. Let us not pay attention to them and notice only those that are accompanied by a comment.

```
DECOMPOSITION {
  DOMAINS  16;
}
INPUT          WORKBENCH;
PHYSICS  HEAT_TRANSFER_3D;
WORKBENCH {
  PATH  cooler_linear.dat;                  # Path to the 3D model file
}
HEAT_TRANSFER_3D {                          # Heat transfer in 3D
  LOAD_STEPS       1;
  MATERIALS {
    ALUMINIUM {                              # Aluminium alloy
      THERMAL_CONDUCTIVITY {
        MODEL  ISOTROPIC;                   # Isotropic model
        KXX          154;                   # Thermal conductivity
      }
    }
```

```
  }
  MATERIAL_SET {
    PART_01  ALUMINIUM;                    # Aluminium alloy set to PART_01
  }
  INITIAL_TEMPERATURE {
    ALL_ELEMENTS  22 + 273.15;             # Uniform initial temperature
  }
  LOAD_STEPS_SETTINGS {
    1 {
      DURATION_TIME   1;
      TYPE  STEADY_STATE;                  #       steady state problem
      MODE        LINEAR;                  # Linear
      SOLVER        FETI;
      FETI {
        PRECONDITIONER       DIRICHLET;
        PRECISION               1E-08;
        MAX_ITERATIONS            200;
        ITERATIVE_SOLVER ORTHOGONALPCG;
        REGULARIZATION       ALGEBRAIC;
      }
      TEMPERATURE {
        BOTTOM_NODE_SET 80 + 273.15;       # Constant temperature
      }
      CONVECTION {                         # Convection
        FACE_SET_01 {                      #           in FACE_SET_01
          HEAT_TRANSFER_COEFFICIENT    80;  # Heat transfer coefficient
          EXTERNAL_TEMPERATURE 22 + 273.15; # Ambient temperature
        }
      }
    }
  }
}
OUTPUT {
  PATH                          results;
  FORMAT                        ENSIGHT;
  RESULTS_STORE_FREQUENCY EVERY_TIMESTEP;
  MONITORS_STORE_FREQUENCY EVERY_TIMESTEP;
  STORE_RESULTS                    ALL;
}
```

Listing 6: ECF for the Cooler example

9.1.2 Configuration via GUI

Default GUI screen, which appears when one runs the program, is shown in Figure 15. It corresponds with the GUI prototype in Figure 7.

We should begin with the loading of 3D model. The cooler's geometry data were created in Ansys Workbench. We can load it in the work flow block on the right. In the **Mesh** tab, we have to change the input format to Ansys and set the path to the file with the geometry. The correct setting is in Figure 16. After one clicks on the **Load** button, the 3D model together with the information about regions are loaded. See the result[4] in Figure 17.

Next, we might define a new material, i.e., aluminium in our case. In the **Data sets** block, we should right click on **materials** and choose **New**. In the newly opened dialog window, we have to specify Name, Description and KXX as it is in Figure 18.

Now, we move to the **Physics** tab. The first data container with the **Property** column can be set straightforward according to Figure 19. In the **Element region properties** container, we have to specify **initial_temperature** for the *ALL_ELEMENTS* region. It works same as materials. One should right click on it and select **New**. A dialog window should appear and the fields inside should be filled same as in Figure 20.

Let us digress from the Cooler example configuration for a while and recall the table and the piecewise function. In the previous dialog window screen shot, we may see the **Type** field. Changing the field value, one might switch to the table or the piecewise function. Example of both is in Figure 21.

The example configuration continuous in the **Materials** tab. The assignment of a material to regions is performed there. In our case, we attach the aluminium to *PART_01* by right clicking on **material_set**.

Another important step is the load step configuration. In the Physics tab, we have leaved the number of load steps set to one. Thus, we have to fill the **Load step 1** tab. The complete configuration may be seen in Figure 22, where the red circles mark the attributes that we should fill in. In the section named **Boundary conditions**, we have to add one region for the **temperature** condition, and second one for the **convection** condition. Again, the add operation can be done via a context menu, which appears when you perform a right click.

This is almost everything necessary before we can export ECF. Nevertheless, the last tab, i.e. **Output**, has not been presented yet. It is the configuration interface for everything that has been introduced in Chapter 2.3. In short, we may define there, what results ESPRESO should produce, and where they should be saved. For the Cooler example, the output configuration would be same as in Figure 23. In Chapter 2.3, we have also discussed the possibility of the

[4]The *BOTTOM_NODE_SET* region is not depicted due to the reasons discussed in Footnote 3.

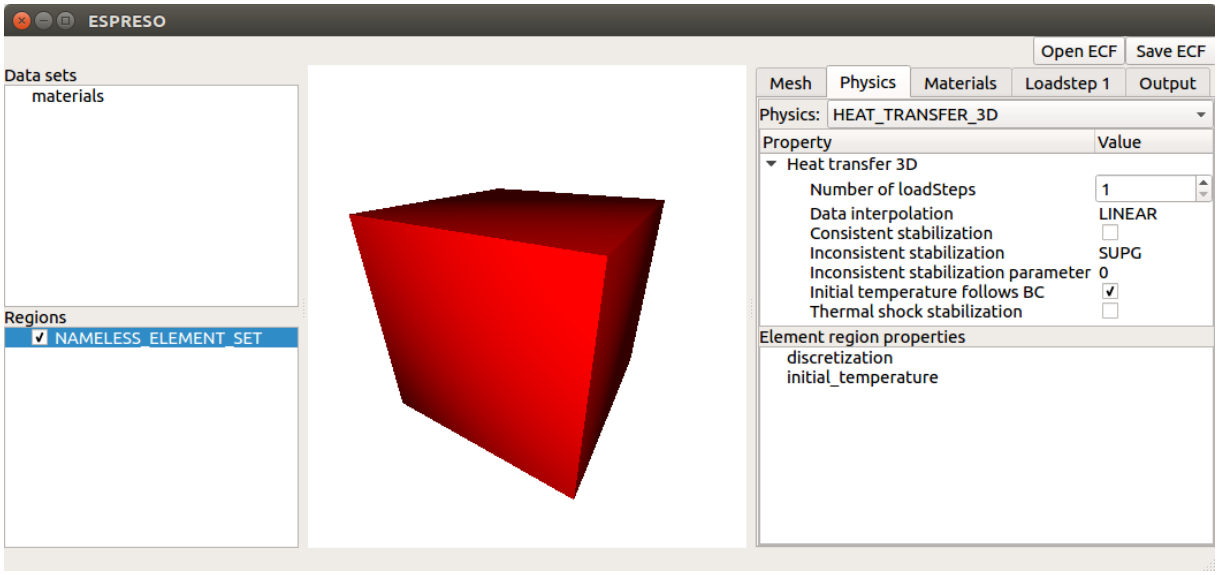

Figure 15: Default GUI screen

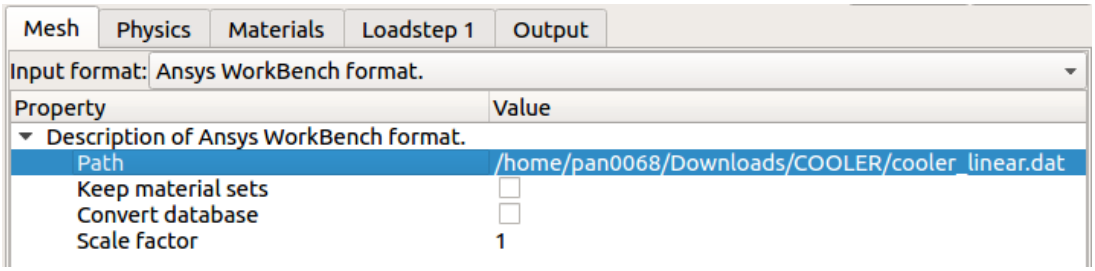

Figure 16: Configuration of the input format

statistical data measurement. **List of monitors** is used for setting these measurements. For example, if we wanted to monitor an average temperature of *FACE_SET_01*, we would add a new monitor via the **Add** button and fill the fields. Figure 24 demonstrates this example.

If we run the solver with the constructed ECF file, we would obtain a graphical result of the heat transfer as shown in Figure 25.

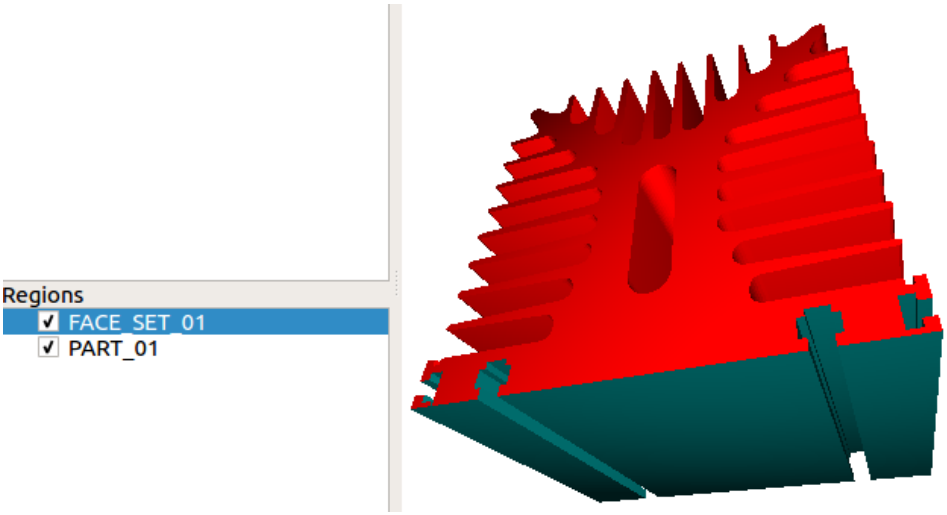

Figure 17: Loaded 3D model with regions

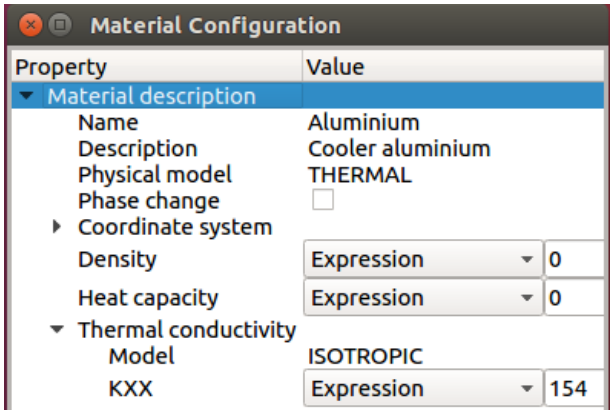

Figure 18: Aluminium - material configuration

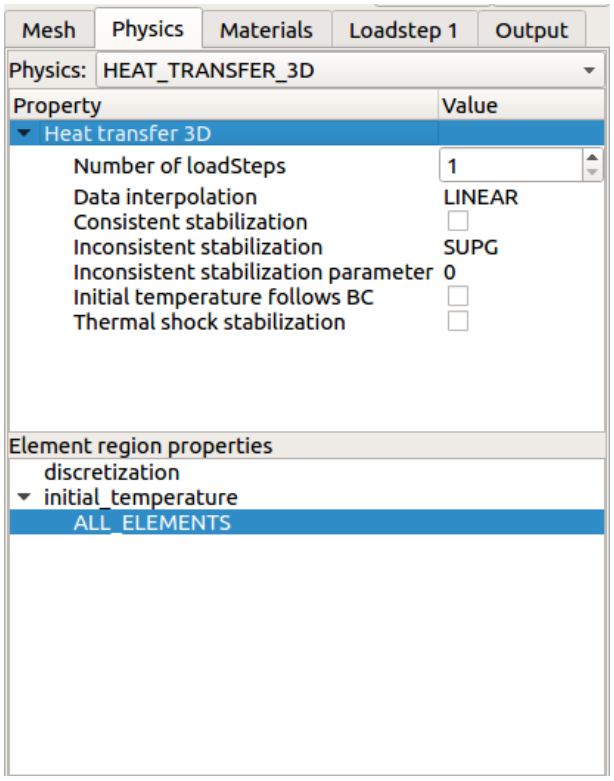

Figure 19: Physics configuration

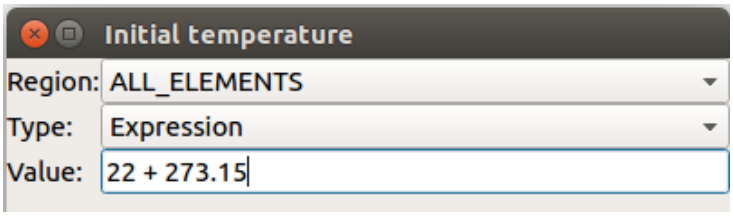

Figure 20: Initial temperature configuration

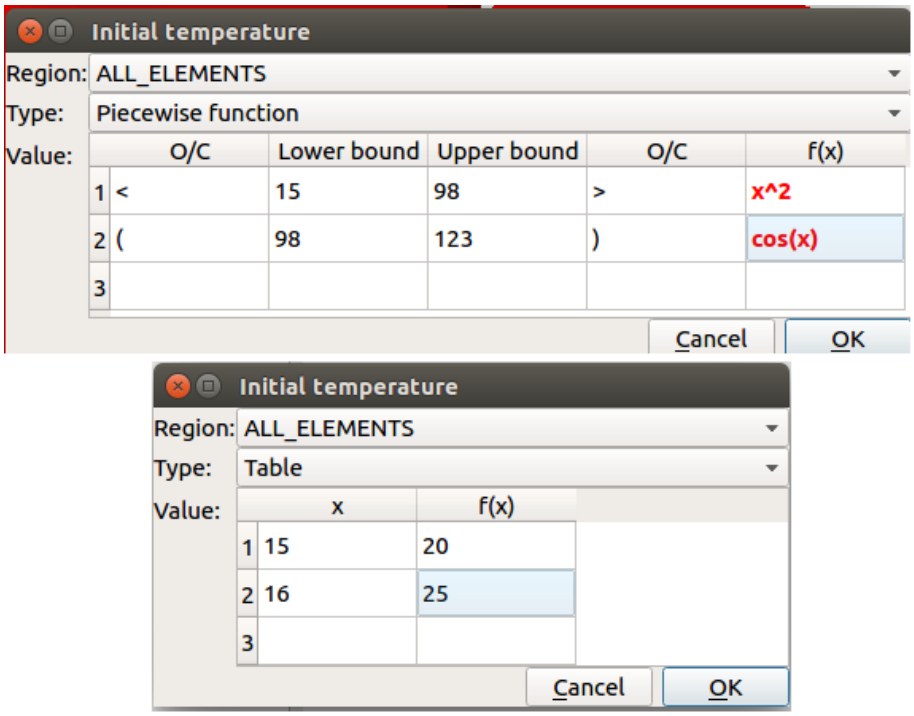

Figure 21: Example of table and piecewise function configuration

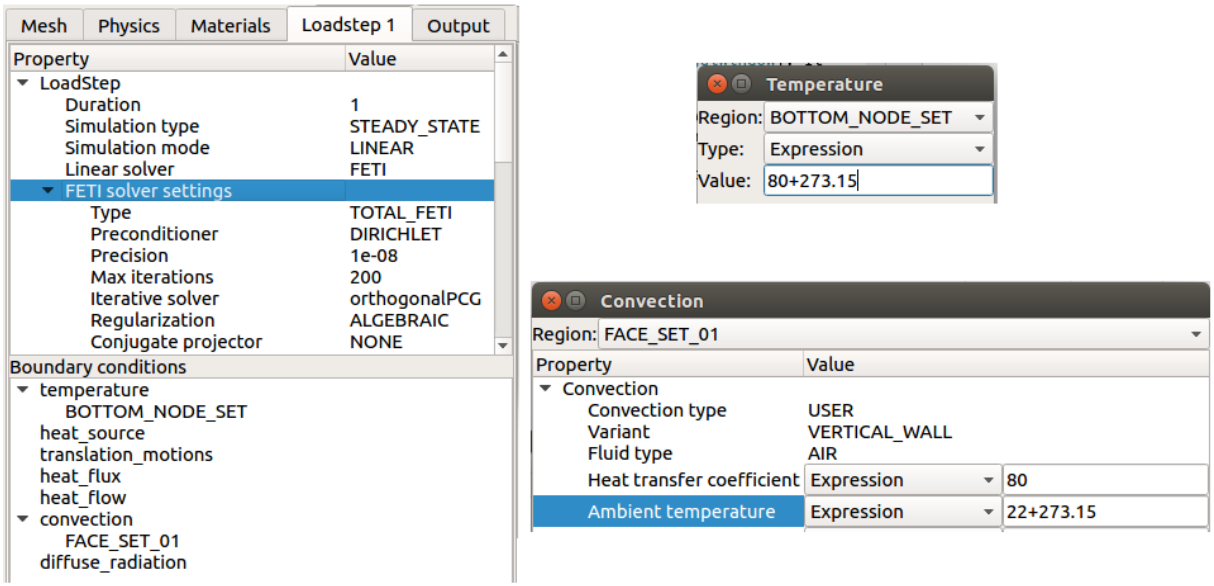

Figure 22: Load step configuration

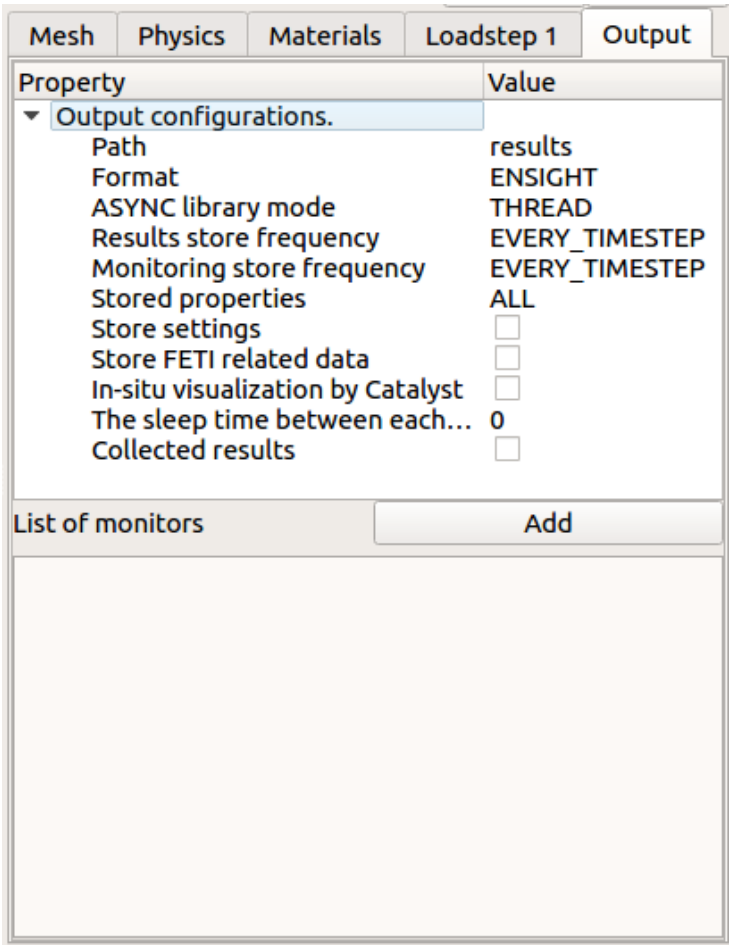

Figure 23: Output configuration

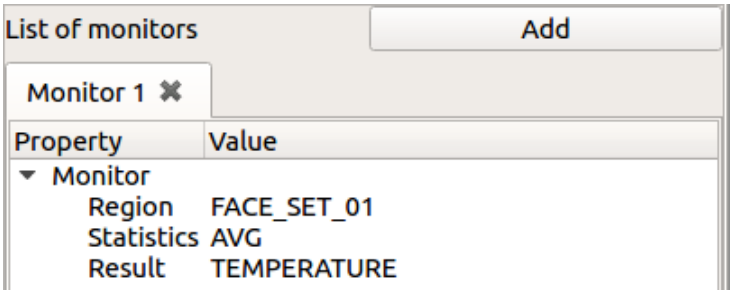

Figure 24: Example of monitor

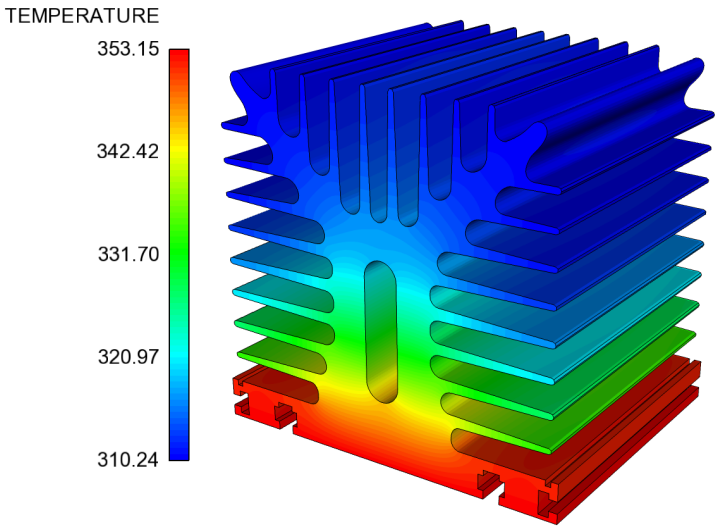

Figure 25: Result of the heat transfer computation

10 Conclusion

At the beginning, the ESPRESO frameworks was presented. The input layer was brought into focus including the description of the ECF files. In Chapters 3 and 4, the requirements and options of the final solution were analysed. In the following chapters, various parts of the new application (GUI) were described and documented. During these chapters, one gets acquainted with many frameworks, libraries, and technologies including Qt, hybrid parallelization with MPI and OpenMP, OpenGL, and ExprTK. All of them were used to create the final application, which was demonstrated in Chapter 9.

The outcome of this thesis is an application, which is directly connected to the ESPRESO solver, and uses its functions for parsing the ECF files. On the basis of the ECF structure, GUI draws appropriate graphical elements via Qt. A change of the ECF structure (i.e. a new parameter added) has automatically an impact on the GUI layout, since the application scans all parameters. The 3D model is rendered via OpenGL using the ESPRESO input layer, which generates a set of triangles representing the rendered object. GUI can be run in parallel via MPI and perform the parallel processing of the ECF files.

In Chapter 8.4, one may find a list of not implemented features in the current version of GUI, which is attached to this thesis. Nevertheless, necessary steps for their implementation were shown. Apart from these functions, one may think of many other improvements. Let us show some examples. The 3D model would deserve better implementation of the zoom and the rotation. Also, a graphical region highlighting with a border or a glare could be considered, when users put the mouse cursor over a region. The current version of GUI uses default visual style of the Qt widgets combining white and gray colours. This combination works, however, it looks too ordinary. A better variation could be pondered together with the layout of GUI elements including spacing between them, borders, font size, font colour, etc. In short, user experience might be improved. Besides the visual aspects, one may try diverse technical possibilities of the GUI and the solver interaction, e.g., the direct execution presented in Chapter 6, or a remote execution with the solver located on a remote server.

To sum up, the final solution is an open source application with GUI featuring 3D rendering with OpenGL. GUI dynamically reacts to the changed in the ECF structure. This thesis includes a complete documentation of the application development including the analysis, the design and the implementation. The application was demonstrated on a basic example of heat transfer. In the paragraph above, the possible future work is summarised. Thus, all requirements specified in the thesis assignment should be fulfilled.

References

[1] Fast Solver for HPC users. *IT4I Espreso* [online]. [cited 2018-02-07]. <http://espreso.it4i.cz/>

[2] CSC - IT CENTER FOR SCIENCE LTD. *Elmer* [online]. [cited 2018-01-29]. <https://www.csc.fi/web/elmer/elmer>

[3] LYLY, Mikko. *ElmerGUI manual v. 0.4* [online]. CSC – IT Center for Science, last revision 22nd May 2017 [cited 2018-01-29]. <http://www.nic.funet.fi/pub/sci/physics/elmer/doc/ElmerguiManual.pdf>

[4] MALINEN, Mika. RABACK, Peter. *Overview of Elmer* [online]. CSC – IT Center for Science, last revision 22nd May 2017 [cited 2018-01-29]. <http://www.nic.funet.fi/pub/sci/physics/elmer/doc/ElmerOverview.pdf>

[5] DHONDT, Guido. WITTIG, Klaus. *CalculiX - A Free Software Three-Dimensional Structural Finite Element Program* [online]. [cited 2018-01-29]. <http://www.dhondt.de/>

[6] WITTIG, Klaus. *CalculiX USER'S MANUAL - CalculiX GraphiX, Version 2.13* - [online]. Last revision 7th October 2017 [cited 2018-01-29]. <http://www.dhondt.de/cgx_2.13.pdf>

[7] Calculix Launcher. *Calculix - Free Finite Element Software* [online]. [cited 2018-01-29]. <http://calculixforwin.blogspot.cz/2015/05/calculix-launcher.html>

[8] OPEN CASCADE. What is SALOME?. *SALOME* [online]. [cited 2018-01-29]. <http://www.salome-platform.org/>

[9] EDF R&D. *Code_Aster - Analysis of Structures and Thermomechanics for Studies & Research* [online]. [cited 2018-01-29]. <https://www.code-aster.org/V2/UPLOAD/DOC/Presentation/plaquette_aster_en.pdf>

[10] EDF R&D. *Presentation of code_aster and Salome-Meca* [online]. [cited 2018-01-29]. <https://www.code-aster.org/V2/UPLOAD/DOC/Formations/01-overview-2.pdf>

[11] GTK+ Features. *The GTK+ Project* [online]. [cited 2018-01-29]. <https://www.gtk.org/features.php>

[12] GTK+ Download: GNU/Linux. *The GTK+ Project* [online]. [cited 2018-01-29]. <https://www.gtk.org/download/linux.php>

[13] GTK+ Download: Windows. *The GTK+ Project* [online]. [cited 2018-01-29]. <https://www.gtk.org/download/windows.php>

[14] GTK+ Download: Mac OS X. *The GTK+ Project* [online]. [cited 2018-01-29]. <https://www.gtk.org/download/macos.php>.

[15] *Glade - A User Interface Designer* [online]. Last revision 8th January 2018 [cited 2018-01-29]. <https://glade.gnome.org>

[16] Overview. *wxWidgets Cross-Platform GUI Library* [online]. [cited 2018-01-29]. <https://www.wxwidgets.org/about/>

[17] Download. *wxWidgets Cross-Platform GUI Library* [online]. [cited 2018-01-29]. <https://www.wxwidgets.org/downloads/>

[18] List of Integrated Development Environments. *wxWiki* [online]. Last revision 2nd January 2015 [cited 2018-01-29]. <https://wiki.wxwidgets.org/List_of_Integrated_Development_Environments>

[19] Get Qt. *Qt* [online]. [cited 2018-01-29]. <https://www.qt.io/download>

[20] Supported Platforms. *Qt Documentation* [online]. [cited 2018-01-29]. <http://doc.qt.io/qt-5/supported-platforms.html>

[21] Qt APIs & Tools, Libraries and Qt Creator IDE. *Qt* [online]. [cited 2018-01-29]. <https://www.qt.io/qt-features-libraries-apis-tools-and-ide/?hsCtaTracking=5132775e-e46e-4d37-a8ef-72ffc974b5b9%7Ca12ac986-23df-4dd4-85e1-b244030056f8>

[22] Create Your First Applications. *Qt Documentation* [online]. [cited 2018-01-31]. <https://doc.qt.io/qt-5.10/gettingstarted.html#create-your-first-applications>

[23] QWidget Class. *Qt Documentation* [online]. [cited 2018-01-31]. http://doc.qt.io/qt-5/qwidget.html

[24] Layout Management. *Qt Documentation* [online]. [cited 2018-01-31]. http://doc.qt.io/qt-5/layout.html

[25] Signals & Slots. *Qt Documentation* [online]. [cited 2018-01-31]. <http://doc.qt.io/qt-5/signalsandslots.html>

[26] Hybrid Parallelism: Parallel Distributed Memory and Shared Memory Computing. *Intel (R) Software Developer Zone* [online]. Last revision 12th May 2016 [cited 2018-01-31]. <https://software.intel.com/en-us/articles/hybrid-parallelism-parallel-distributed-memory-and-shared-memory-computing>

[27] Set up the Environment. Installation of the library ESPRESO. *ESPRESO 0.9 documentation* [online]. [cited 2018-01-31]. <http://espreso.it4i.cz/doc/installation.html#set-up-the-environment>

[28] OpenMP Compilers & Tools. *OpenMP* [online]. [cited 2018-01-31]. <http://www.openmp.org/resources/openmp-compilers/>

[29] BARNEY, Blaise. *Message Passing Interface (MPI)* [online]. Lawrence Livermore National Laboratory, last revision 20th June 2017 [cited 2018-01-31]. <https://computing.llnl.gov/tutorials/mpi/#LLNL>

[30] DE VIRES, Joey. OpenGL. *Learn OpenGL* [online]. [cited 2018-01-31]. <https://learnopengl.com/Getting-started/OpenGL>

[31] DE VIRES, Joey. *Learn OpenGL* [online]. [cited 2018-01-31]. <https://learnopengl.com/>

[32] DE VIRES, Joey. Hello Triangle. *Learn OpenGL* [online]. [cited 2018-01-31]. <https://learnopengl.com/Getting-started/Hello-Triangle>

[33] HAWKINS, Kevin. a Dave. ASTLE. *OpenGL game programming*. Roseville, CA: Prima Tech, 2001. ISBN 0-7615-3330-3.

[34] OpenGL and OpenGL ES Integration. Qt GUI. *Qt Documentation* [online]. [cited 2018-01-31]. <http://doc.qt.io/qt-5/qtgui-index.html#opengl-and-opengl-es-integration>

[35] QOpenGLShaderProgram Class. *Qt Documentation* [online]. [cited 2018-01-31]. <http://doc.qt.io/qt-5/qopenglshaderprogram.html>

[36] QOpenGLWidget Class. *Qt Documentation* [online]. [cited 2018-01-31]. <http://doc.qt.io/qt-5/qopenglwidget.html>

[37] DE VIRES, Joey. Camera. *Learn OpenGL* [online]. [cited 2018-01-31]. <https://learnopengl.com/Getting-started/Camera>

[38] OpenGL Picking Tutorial. *Lighthouse3d.com* [online]. [cited 2018-01-31]. <http://www.lighthouse3d.com/tutorials/opengl-selection-tutorial/>

[39] PARTOW, Arash. *C++ Mathematical Expression Library* [online]. [cited 2018-02-07]. <http://partow.net/programming/exprtk/>

[40] QLineEdit class. *Qt Documentation* [online]. [cited 2018-02-07]. <http://doc.qt.io/qt-5/qlineedit.html>

[41] QItemDelegate class. *Qt Documentation* [online]. [cited 2018-02-07]. <http://doc.qt.io/qt-5/qitemdelegate.html>

[42] QRegularExpression class. *Qt Documentation* [online]. [cited 2018-02-07]. <http://doc.qt.io/qt-5/qregularexpression.html>

[43] QValidator class. *Qt Documentation* [online]. [cited 2018-02-07]. <http://doc.qt.io/qt-5/qvalidator.html>

[44] MACIEIRA, Thiago. Linux-icc: always compile applications as position-independent execs. *Qt Code Review* [online]. Last revision 26th January 2017 [cited 2018-02-13]. <https://codereview.qt-project.org/#/c/183454/>

[45] New Qt5 signal/slot syntax not working with Intel Compiler 16. *Qt Bug Tracker* [online]. Last revision 11th May 2017 [cited 2018-02-13]. <https://bugreports.qt.io/browse/QTBUG-52439>

[46] Modern OpenGL. *GLUMPY* [online]. [cited 2018-02-20]. <https://glumpy.github.io/modern-gl.html>

[47] Primitive. *OpenGL Wiki* [online]. Last revision 31st December 2017 [cited 2018-04-06]. <https://www.khronos.org/opengl/wiki/Primitive>

[48] SALOME Desktop. *Salome Platform Documentation* [online]. [cited 2018-04-20]. <http://docs.salome-platform.org/latest/gui/GUI/salome_desktop_page.html>

A Appendix on CD

Files

- *cooler.dat* - 3D mesh with the Cooler example in the Ansys Workbench format.
- *espreso.zip* - a ZIP archive with the Git repository of ESPRESO including GUI. For this thesis, the important subdirectory is *espreso/src/gui*, where the main source code of GUI is located.
- *guide.pdf* - installation guide of ESPRESO.
- *thesis.pdf* - PDF copy of this thesis.

B Figures of existing GUI configuration tools

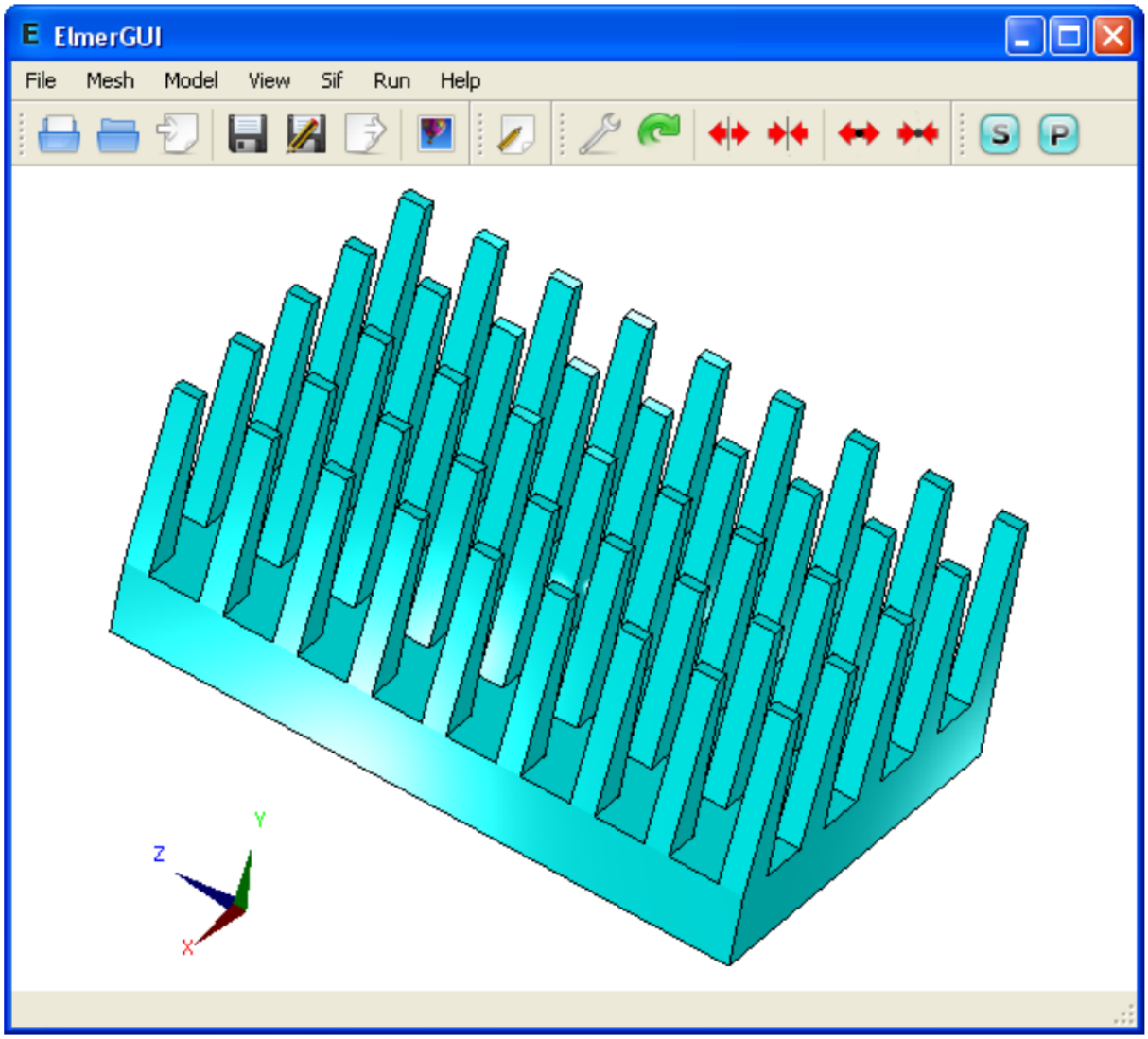

Figure 26: ElmerGUI [3]

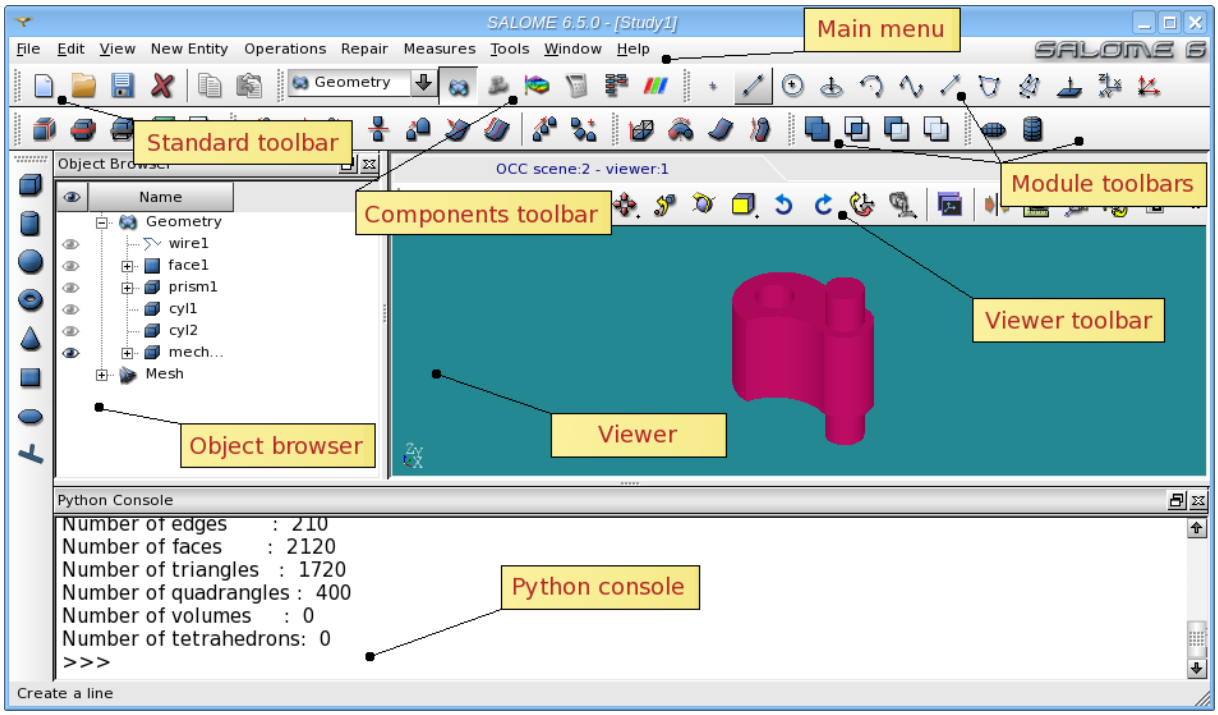

Figure 27: SALOME [48]

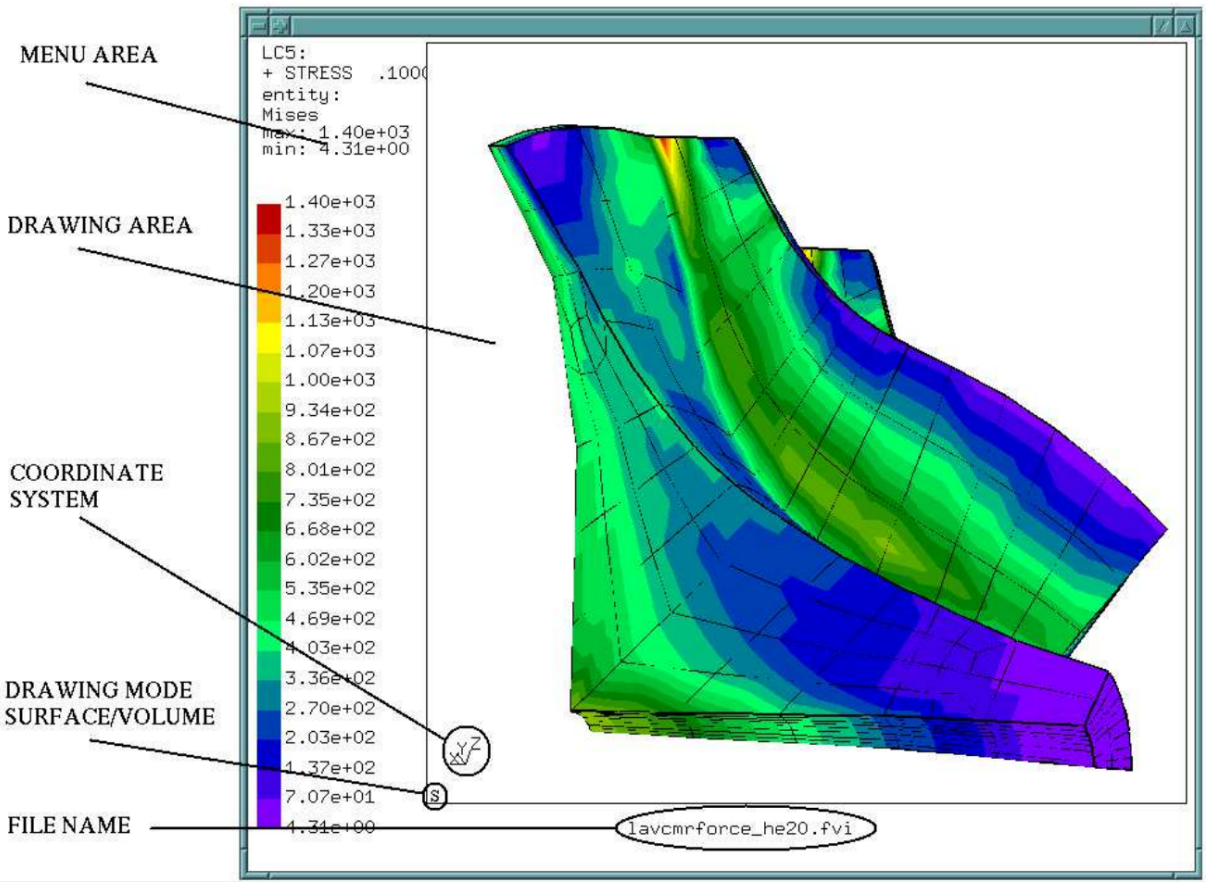

Figure 28: CalculiX GraphiX [6]