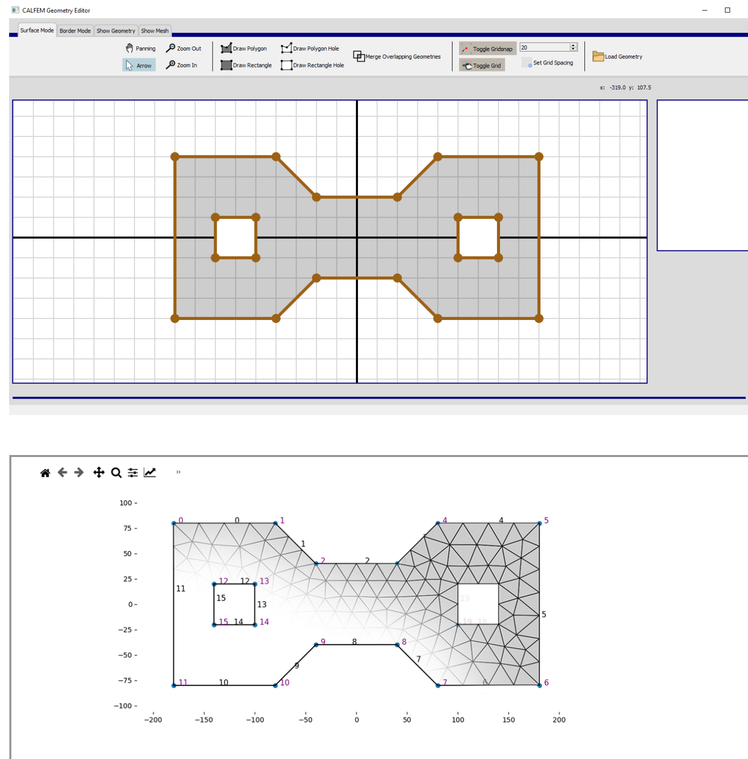
LUND UNIVERSITY



CALFEM GEOMETRY EDITOR

Implementing an interactive geometry editor for CALFEM

KARL ERIKSSON

Structural Mechanics

Master's Dissertation

DEPARTMENT OF CONSTRUCTION SCIENCES

DIVISION OF STRUCTURAL MECHANICS

ISRN LUTVDG/TVSM--21/5255--SE (1-42) | ISSN 0281-6679

MASTER'S DISSERTATION

CALFEM GEOMETRY EDITOR

Implementing an interactive geometry editor for CALFEM

KARL ERIKSSON

Supervisor: Dr **JONAS LINDEMANN**, Division of Structural Mechanics, LTH | Lunarc.
Assistant Supervisor: **KARIN FORSMAN**, Lic Eng, Division of Structural Mechanics, LTH.

Examiner: Professor **OLA DAHLBLOM**, Division of Structural Mechanics, LTH.

Copyright © 2021 Division of Structural Mechanics,
Faculty of Engineering LTH, Lund University, Sweden.

Printed by V-husets tryckeri LTH, Lund, Sweden, September 2021 *(Pl)*.

For information, address:
Division of Structural Mechanics,
Faculty of Engineering LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: www.byggmek.lth.se

Abstract

Commercial finite element codes are designed with a wide variety of tools and options to suit the needs of many different applications. However, in a teaching setting this is not always desirable as the effort required to learn the software draws focus and time from the teaching of the theoretical concepts. Using fully developed solvers might also allow solving of exercises without fully understanding the underlying concepts as the software handles it for you. To address this, CALFEM (Computer Aided Learning of the Finite Element Method) was developed at LTH. CALFEM provides a toolbox to solve finite element problems but with a clear connection between the methods used and the governing theoretical concepts. Today, CALFEM is fully implemented in MATLAB but since a few years back a Python version is under development as well. In the finite element method a fundamental part of the problem is the geometry, which is used to generate the finite element mesh. As of now, in order to produce a geometry in CALFEM for Python, all points and connecting lines and surfaces must be manually defined in the code. This is both tedious work and limiting to a simple geometry with few points before becoming difficult to handle. Because of this, students may struggle and spend much time working with defining the geometrical points and connections correctly. To address this issue, the purpose of this dissertation has been to implement an interactive geometry editor designed to create simple geometries in an intuitive way.

The editor has been developed in Python combined with a graphical user interface designed in PyQt5. The implemented functions allow creation and editing of polygon and rectangular surfaces in a QGraphicsScene. The geometrical objects created in the QGraphicsScene can then be transformed into a CALFEM geometry object. From this stage a CALFEM mesh can be generated and exported. The main functionality of the editor is developed towards use in Python but option has been included to export mesh results to MATLAB as well. To achieve the goal of creating a simple and intuitive program the user interface was implemented using a ribbon like design to handle the controls. The functionality in the program window was based on the idea of direct manipulation.

Acknowledgements

The work in this master's dissertation was carried out at the Division of Structural Mechanics at LTH. I would like to thank my supervisors Jonas Lindemann and Karin Forsman for their guiding and support during the course of the project and for providing valuable means of communication among the restrictions during the Covid-19 pandemic.

August 2021, Lund.

Contents

Abstract **I**

Acknowledgements **III**

Table of Contents **VI**

1 Introduction **1**
1.1 Background 1
1.1.1 CALFEM 1
1.1.2 Geometry and Mesh Creation 1
1.2 Aim 2
1.3 Method 2
1.4 Limitations 3
1.4.1 Two Dimensional Geometries 3
1.4.2 Boundary Conditions 3
1.4.3 Geometrical Shapes 3

2 Review of Existing Tools **5**
2.1 PowerPoint 5
2.2 Pdetool 5
2.3 Gmsh GUI 7

3 CALFEM Geometry Editor Overview **9**
3.1 User Interface 9
3.2 Class Description 10
3.3 Geometry Modeling 11

4 CALFEM Geometry Editor Implementation **13**
4.1 Underlying Libraries 13
4.1.1 CALFEM and Gmsh 13
4.1.2 PyQt and Qt 14
4.2 Core Functions 15
4.2.1 Converting Graphical Objects to a Gmsh Geometry Description 15
4.2.2 Snapping of Points to Points and Snapping of Points to Edges . 16
4.2.3 Views and Modeling Modes 17
4.2.4 Export/Import Files 17
4.3 Graphical User Interface 18

4.4 Editor Code Interaction 19

5 Discussion and Conclusions **21**
5.1 In-Work Decisions 21
5.1.1 Surfaces and Geometrical Shapes vs Nodes and Connections 21
5.1.2 Drawing Using Only Polygons 22
5.2 Conclusions 22
5.3 Future Work 23
5.3.1 Improvement on Existing Functions 23
5.3.2 Extended Functionality 23
5.3.3 Computational Efficiency 24

Bibliography **25**

A Manual **27**

1 Introduction

In this dissertation a program has been developed to create and modify geometries to be used in the Finite Element Method (FEM) library CALFEM for Python [1]. This report covers the work of the project starting with the basis of why the program is needed and what purpose it should serve. The reader is expected to have basic knowledge of program development and the finite element method. This chapter describes the methodology of how the work was performed and which limitations were set for the project.

1.1 Background

1.1.1 CALFEM

CALFEM (Computer Aided Learning of the Final Element Method) [2] is a library developed at Lund University with the purpose of helping the teaching of the finite element method at the University. The idea of the library is to provide a toolbox for solving finite element problems in a pedagogical setting, providing a clear connection between the theoretical concepts of the finite element model and the methods used when programming [3]. The development started already in the 70's and has consisted of multiple programming languages. The core of CALFEM today is designed in MATLAB [4] but since a few years back a Python version (cloned from the existing MATLAB version) has been developed as well [5]. There are many benefits of using Python [6], one of them is that Python is open source and does not require any expensive licenses contrary to MATLAB. Moreover the syntax of Python forces the user to write proper readable code. One advantage of using MATLAB is easy use of matrices and matrix operations, which is of great importance in the finite element method which is based on the use of matrices. However, since the launch of NumPy for Python [7] similar tools are available in Python.

1.1.2 Geometry and Mesh Creation

An important step in the finite element method is the generation of a finite element mesh. In order to produce a mesh the geometry must first be defined in space and assigned with boundary conditions. In CALFEM there is no user interface which enables the user to create and modify geometries in an effective and intuitive way, the points and lines of the geometry must instead be defined manually in the code. For CALFEM in Python a mesh class has been developed [8] using Gmsh [9] in order to perform the meshing but the geometry must be defined manually in the code which is a cumbersome approach. Manually producing the geometries introduces unnecessary complexity for certain exercises in the FEM courses and may draw focus from other

more important key aspects of teaching the basics of the finite element method. This calls for the need of a graphical user interface to allow students to, in a simple way, create geometries needed in the exercises. Using a graphical user interface enables a different working process compared to only relying on scripting, most notably it provides direct input when drawing geometries.

To fulfill the need of a graphical user interface for creating geometries a third-party software could be used, for example the mesh and geometry generator from the library pdetool [10] may be used in MATLAB. However, the drawback with these software is that they often are created as professional programs geared towards engineers in finite element analysis with a vast number of features and tools and with a focus on efficiency and flexibility. This usually means that the programs have a high learning curve and the user must spend a lot of time learning the program before being able to use it properly. For the students to use these types of software would be very time consuming for very little benefit.

1.2 Aim

The aim of the project is to develop a Graphical User Interface (GUI) with tools to draw 2D geometries, integrated with Gmsh and CALFEM for Python. As further requirements the GUI should be easy and intuitive to use without the need of extensive training and should be directed towards an educational setting on university level. The program should also be versatile to be used in three different scenarios:

1. As a function in CALFEM for Python where the editor is opened and allows creation or editing of geometries before closing and continuing with the script. To allow for interactivity pre-existing geometries should be able to be passed to the function with the option of editing.
2. As a stand alone program with possibilities of drawing geometries, add boundary conditions and generating mesh results that can be saved for further use in both CALFEM for MATLAB and CALFEM for Python.
3. As a component that may be integrated in other Qt-based graphical interfaces to be used as a resource in courses at the university or in future programs.

1.3 Method

The initial idea of the program was to use the QGraphicsScene [11] library from PyQt5 [12] and use the different available shapes: lines, ellipses, rectangles and polygons. The implementations of these item classes would be used to draw and create the geometry as well as tracking the components in the QGraphicsScene.

A number of different software packages were explored and compared to evaluate which features and representations were deemed to be relevant to use as inspiration.

A baseline was laid down of the main functionality of the program. Furthermore, limitations were added to limit the scope of the work and ensure the simplicity of the program. The work then started by implementing the required functionality. As the work progressed, recurring meetings were held discussing the state of the project and how to proceed, creating an iterative process. Once the core of the program was finished, the graphical user interface was restructured and designed with a purpose of user friendliness and simplicity in mind.

1.4 Limitations

During the course of the project some limitations had to be made to reduce the scope of the work in order to fit it within the time frame available. Some limitations were also included to ensure that the program remained simple and easy to use.

1.4.1 Two Dimensional Geometries

As the introductory FEM courses at the university mainly deal with two dimensional problems the program was limited to two dimensions. Adding depth to create three dimensional geometries greatly increases both the complexity of the drawing program and complicate the overview of the problem for the user. Thus, limiting it to two dimensions reduces the complexity of the usage of the program.

1.4.2 Boundary Conditions

In the finite element method, a key component to produce accurate models are the boundary conditions. When creating geometries in mechanics examples of these conditions are roller support and fixed support. However, the program is meant to be general-purpose and be able to handle other problem areas such as heat transfer and groundwater flow. Each of these areas have differences such as numbers of degrees of freedom and units of measure. Therefore, it was determined that implementing an interface covering all these possibilities would be too complex. Instead, the application extends the functionality in CALFEM for Python, by allowing the user to set markers on the target edges and vertices. The markers are then saved together with the geometry to allow the user to handle the boundary conditions later in the code. This is also in line with the principles of CALFEM, allowing the user to have a better interpretation of the implementation [3]. Furthermore, it eliminates the need of using the program to adjust the values of the conditions which can make it easier to adjust and try different values.

1.4.3 Geometrical Shapes

The initial idea was to use the four geometrical shapes: lines, ellipses, rectangles and polygons. However, due to decisions made during the progress, the implementation

was limited to only using polygons and rectangle as a special case of a polygon. For further discussion see Section 5.1.2.

2 Review of Existing Tools

To implement a program with the purpose of creating geometries and integrating the necessary components to be used in a FEM setting, some inspiration was obtained by looking at other similar programs. This to gain knowledge and ideas of what features and tools from established applications that might be good to use and are easy to understand in an intuitive way, but also to find what should be avoided and what could be made better or simpler.

2.1 PowerPoint

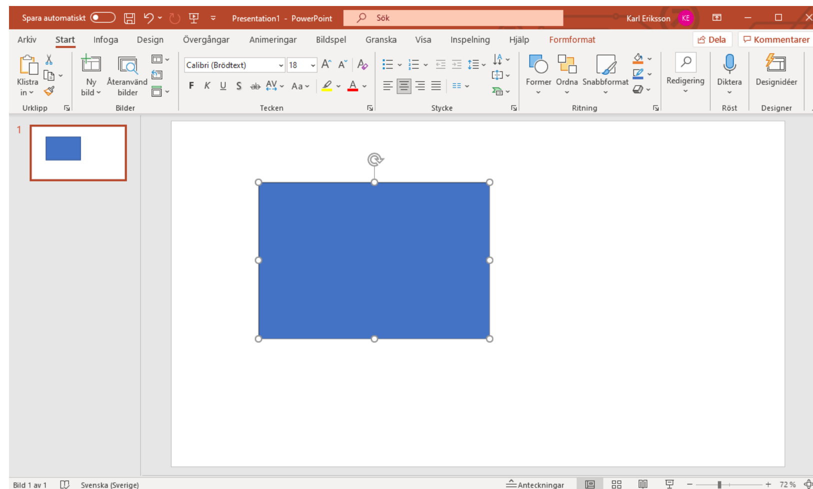


Figure 2.1: Snapshot example from working with geometries in PowerPoint

PowerPoint [13] was used as a source of inspiration of how to draw surfaces in an easy to use and intuitive way. PowerPoint is a tool for making presentations, shown in Figure 2.1, with drawing geometries only as a small sub-tool. Therefore, it serves as a good viewpoint of how to make a simple and intuitive drawing tool since the user is not expected to have any previous experience. How the basic shapes, such as rectangles and circles, are created in the program were explored. Also, how these shapes are handled when moving, scaling and reshaping were examined. The findings were used as ideas of how to implement similar functionality.

2.2 Pdetool

Pdetool is a part of the Partial Differential Equation (PDE) Toolbox in MATLAB [10] and operates as a stand-alone program launched from the MATLAB command line. The program serves as a full solver of numerous PDE problems, but in this

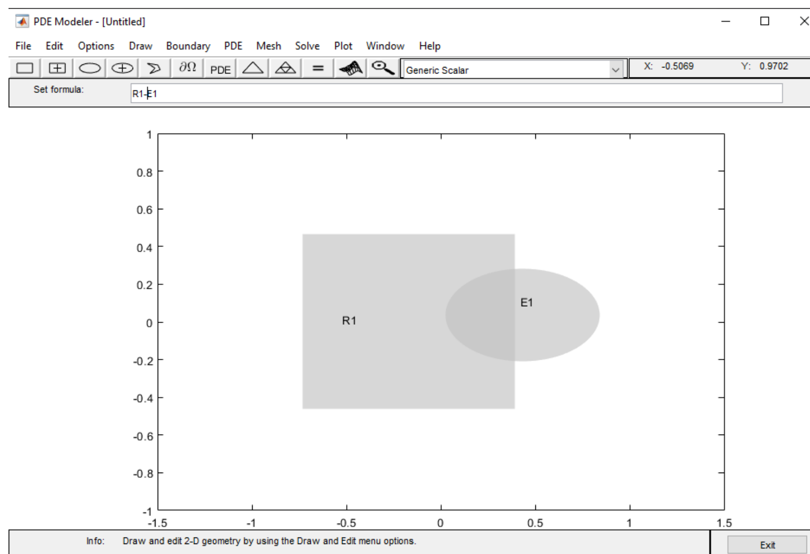


Figure 2.2: Snapshot example from working with geometries in Pdetool

work only the tools of creating geometries and meshes was considered when exploring its functionalities. Pdetool is used as a supplement to the CALFEM for MATLAB library for creating geometries and meshes in some FEM courses at Lund University. The program is defined for a 2D domain and the geometry creator, shown in Figure 2.2, is based on combining simple graphical objects. The available shapes are: rectangle, circle, ellipse and polygon. These shapes are moved and rotated to form overlapping groups. Each shape is given a variable and a global formula governs how the shapes should combine to form the geometry; A positive sign adds the shape to the combined geometry while a negative sign subtracts the shape yielding a hole or a cut depending on the overlap.

Pros:

- With the formula it is very easy to create holes and cut out parts of an object.
- The very basic tools are presented in a simple way and more advanced options for e.g. meshing is available through the menu, but not something the inexperienced user has to worry about.
- The use of right-clicking to finish a polygon is helpful, not having to rely on user accuracy to click near the starting point to close the surface.

Cons:

- The grouping and selection of items can be confusing, e.g. when clicking on overlapping geometries everything behind the cursor is selected. When clicking again the new selection depends on whether the new selection involves any of the already selected items or not. To completely reset the selection the background has to be clicked first.

- When toggling between the different views there is no clear indication of which view you are currently in (except graphical effects on the geometry, but no title or similar).
- Formula may sometimes be experienced as inconsistent i.e. $(R1 + R2) - R3$ is not the same as $R1 + R2 - R3$
- Using only overlapping geometries is intuitive but might require a lot of effort to realize a good way of constructing a more complex geometry. For complex geometries it might require a large number of items which all overlap. This makes it more difficult to get a proper overview with all the overlapping geometries. It may also make it more difficult to select the correct element/elements with the way selecting works.
- Tooltip info is good but placed in the bottom left corner which makes it less noticeable as the focus usually is around the drawing canvas and the toolbar.

2.3 Gmsh GUI

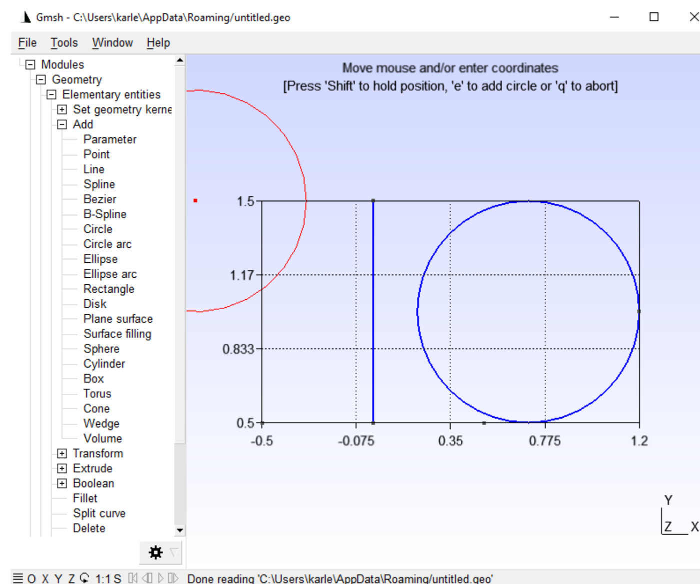


Figure 2.3: Snapshot example from working with geometries in the Gmsh GUI

Gmsh [9], which is the meshing library used in CALFEM. It has its own GUI, shown in Figure 2.3, but upon inspection there are some drawbacks. The GUI is not very intuitive for non-experienced users and proves difficult to handle without first having to learn the GUI in detail. Second, it is designed to handle 3D geometries which in turn complicates the usage of simple 2D geometries. Furthermore, it is built around usage of the full Gmsh library, i.e. a vast amount of different options and functions are included, which complicates the understanding for an inexperienced user even more.

3 CALFEM Geometry Editor Overview

This chapter gives a brief overview of the structure of the CALFEM Geometry Editor by describing the underlying classes and how they interact. The geometry modeling is described to explain the handling of the geometrical items drawn in PyQt5 and the connection to the geometry generated using CALFEM. Furthermore the design is summarized by describing the user interface controls and how they govern the functions used. For more extensive descriptions of the user interface and commands see Appendix A.

3.1 User Interface

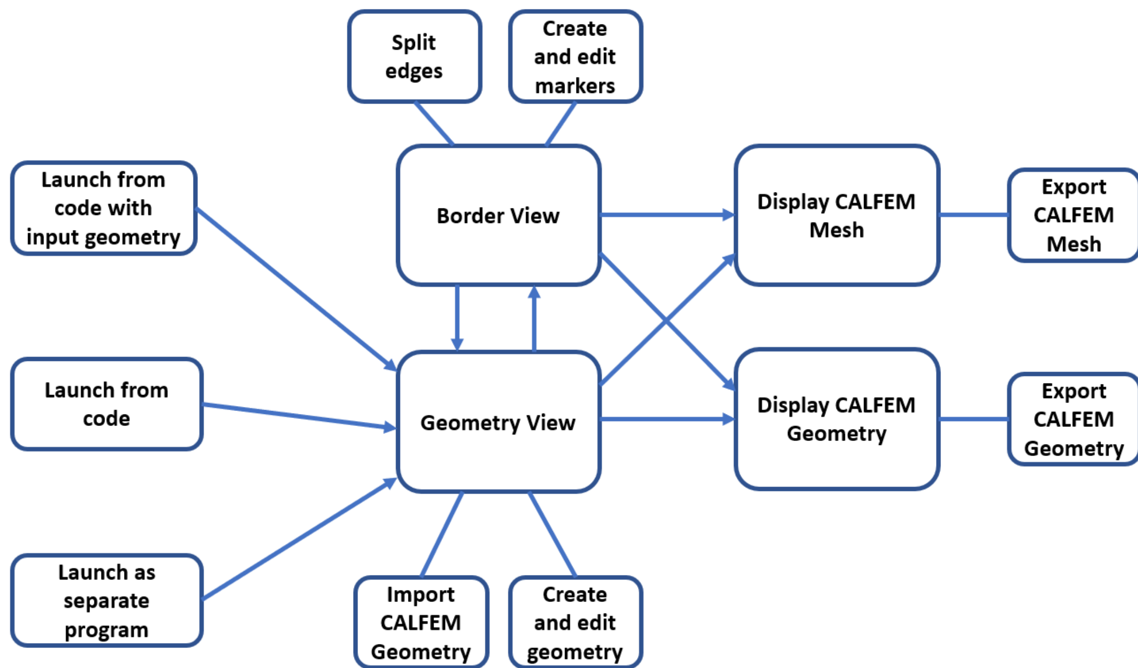


Figure 3.1: Diagram of the connections between the different views and tools in the user interface

Depicted in Figure 3.1 is the work flow of the views and tools in the program. The program can be launched with three different options: from command, from command with a pre-existing geometry as argument and as a stand-alone program. The program is always launched in the geometry editing view, Figure 3.2 shows an example of the program launched without any pre-existing geometry. From there it is possible, by selecting tools in the ribbon, to load geometries from file, add geometries by drawing or edit existing geometry objects. Pressing a tab changes the mode of the program

and automatically changes the available tools for the selected mode. The other editing options available are found in the border view, in which only the borders of the graphical objects are displayed. There, markers may be added to lines or points and lines can be split to add extra nodes along an existing line. When the creation and editing of a geometry is finished it may be generated and displayed as a CALFEM geometry object or as a CALFEM mesh object. If the mesh or geometry is satisfying, they may be exported or will be returned in code depending on the command used to launch the program.

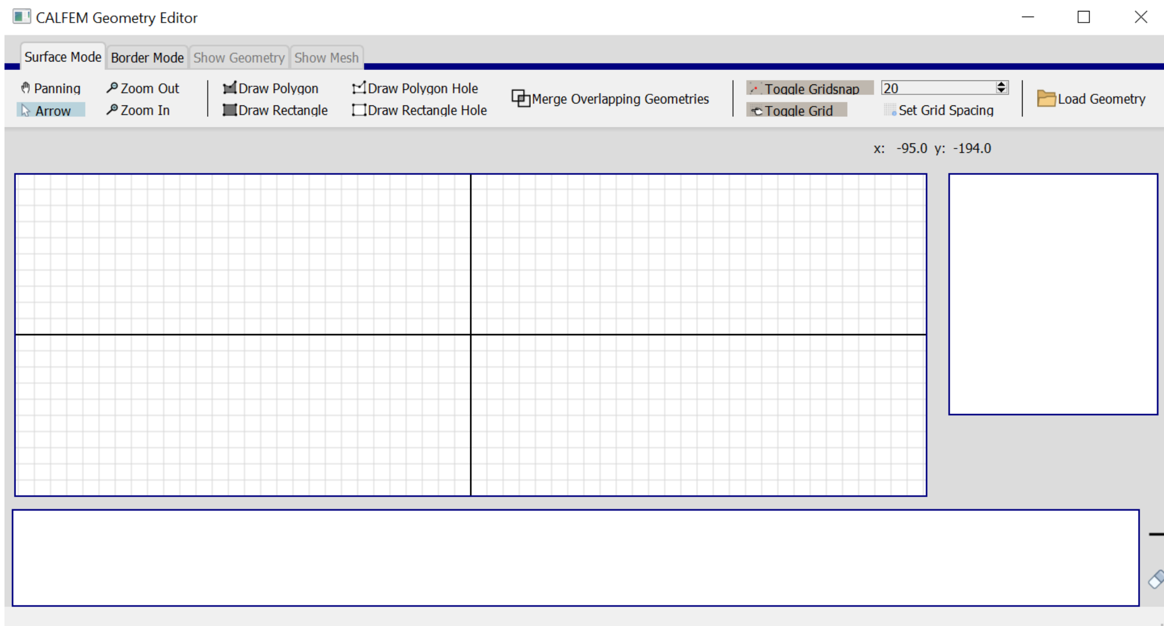


Figure 3.2: Snapshot of the CALFEM Geometry Editor program at launch, before any actions have been performed

3.2 Class Description

Essentially the program is implemented by three different classes: EditorWindow, EditorScene and CALFEM for Python, seen in Figure 3.3. EditorWindow extends the structure natively called MainWindow in PyQt5, similarly EditorScene extends QGraphicsScene from PyQt5. Import and Export are collected sets of methods to handle saving and loading data to the program or scripts. Export is implemented using the Python library pickle [14] to store the desired data in an appropriate file, with the exception of saving to MATLAB which uses the ”savemat” function from SciPy [15]. Import is then structured to unpack the pickle-file in the corresponding variable order and type as the export. The MainWindow and the GraphicsScene are both implemented using PyQt5 together with a large number of methods to handle the creation and editing of the geometries. They are closely intertwined as the MainWindow handles the controls in which methods are performed in the GraphicsScene. In turn the MainWindow is updated depending on actions performed in the GraphicsScene. The user created geometry in the GraphicsScene is then processed to recreate the geometry as a CALFEM geometry object. This geometry object may then be used as an

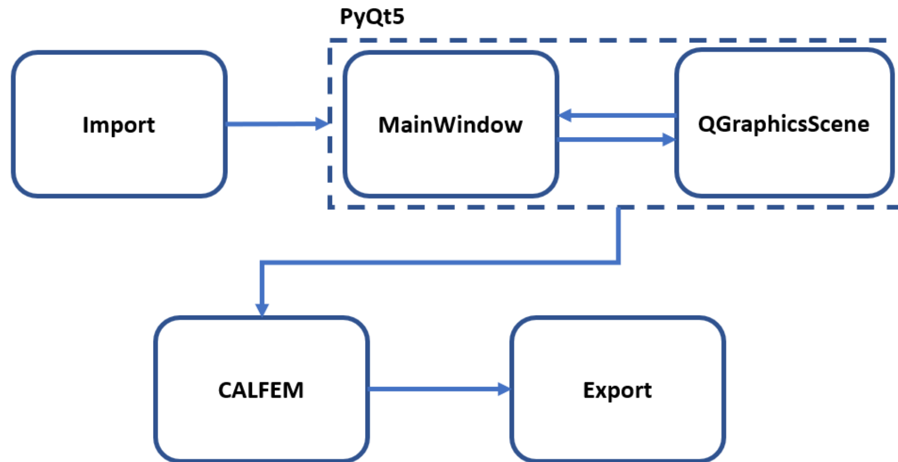


Figure 3.3: Diagram of the classes used in the program

input to the CALFEM mesh class to generate a mesh object and corresponding data arrays.

3.3 Geometry Modeling

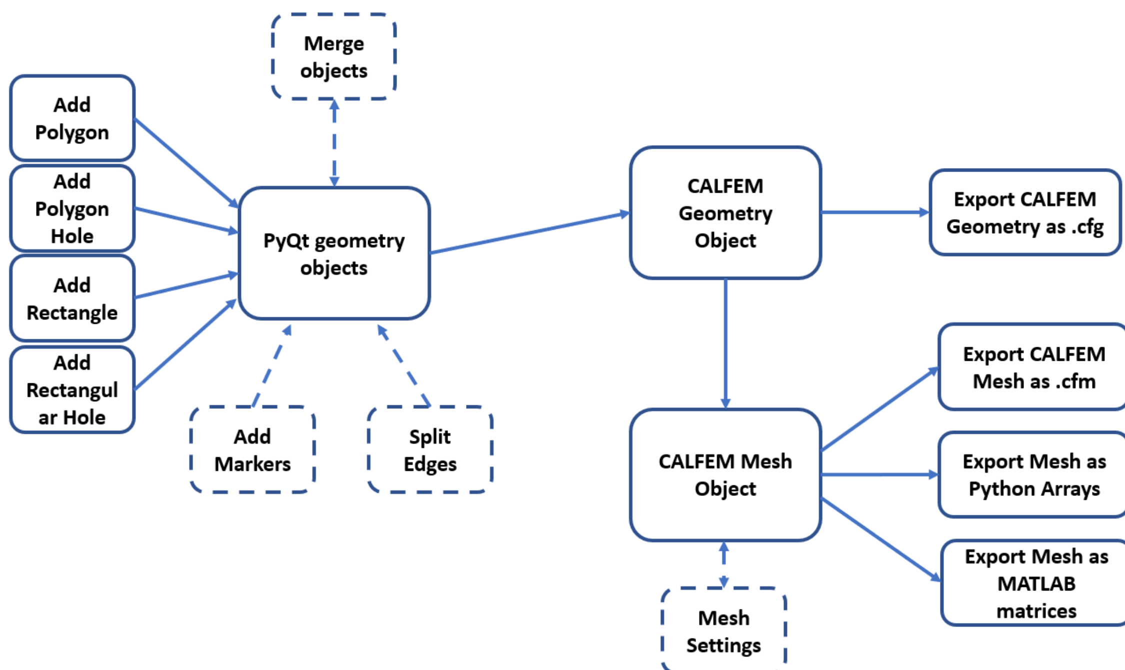


Figure 3.4: How the handling of the geometry objects is performed in the program, dashed arrows show actions that does not add new objects but alter already existing ones

The geometrical objects created by the user in the GraphicsScene are done so using PyQt5 functions and to some extent handled by the corresponding native functions.

However, some attributes had to be manually created to track certain, for the application necessary, variables. Correspondingly some methods in the GraphicsScene such as mouse clicking and moving are overridden in the class to handle certain events. Nevertheless, the objects created and edited by the user in the scene are PyQt5 derived objects.

To create a mesh from the geometry in the scene it must be transformed from PyQt5 objects to the structure used to describe geometries in CALFEM for Python. This is done by creating a CALFEM geometry instance and then iterate through all the objects created in the GraphicsScene. All objects are converted to the proper CALFEM Geometry object structure. It is then possible to save the generated geometry or use CALFEM functions to generate a mesh from this geometry.

4 CALFEM Geometry Editor Implementation

The implementation of the CALFEM Geometry Editor is detailed in this chapter. The underlying libraries used in the implementation are described. Since describing all functions in the program would be too extensive, a few core functions have been selected. These core functions are considered to represent the most important aspects of the implementation and require explanation to provide sufficient insight. Furthermore, the thoughts behind the designing of the GUI are explained.

4.1 Underlying Libraries

4.1.1 CALFEM and Gmsh

In general a series of steps are performed in order to perform simple FEM calculations. These steps consist of [5]:

1. Define the model
2. Generate element matrices
3. Assemble element matrices into the global system of equations
4. Solve the global system of equations
5. Evaluate element forces

Step 1. , 2. and 3. are the steps considered when dealing with the geometry and mesh generation. The geometry must be defined in order to generate a mesh. The mesh then holds all element and the corresponding matrices. These matrices also provide the structure in which the elements are connected to each other to form the geometry as well as the boundary conditions. In CALFEM, the meshing is performed with the Gmsh library. Gmsh [9] is a fast and robust open source finite element mesh generator, available in multiple programming languages including Python. The geometry input to the CALFEM mesh class is therefore based on the geometry handling of Gmsh. Returned when producing a mesh are a series of arrays and dictionaries:

- Dof - Array of all total degrees of freedom that are in the geometry, each row in the array represents a node and its corresponding degrees of freedom make up the columns. The array dimension is hence [number of nodes x degrees of freedom per node].

- Coords - Array containing the coordinates for each degree of freedom, it consists of two or three columns depending on if the geometry is 2D or 3D. In this work only 2D is considered and the array dimension [number of elements x 2].

- Edof - Element degrees of freedom connects each element to its nodes and their degrees of freedom. Number of columns depends on both element type and number of degrees of freedom per node, i.e. standard case with triangular element with two degrees of freedom per node gives a width of 6 (three nodes per element with two degrees of freedom each). The general formula for the array dimension is [number of elements x (nodes per element x degrees of freedom per node)].

- Bdof - Dictionary of all degrees of freedom along the boundaries. Natively all degrees of freedom are assigned in a list to the dictionary entry 0 if marked borders are provided. If a border edge or point is assigned a marker, the marker name of that point or edge is added as a key in the dictionary and a list of the corresponding degrees of freedom as values.

- Element Markers - List where every entry corresponds to the element marker of the element of that index number

- Boundary Elements - Optional argument which provides a dictionary of all elements along the borders

4.1.2 PyQt and Qt

Qt is a graphical user design library based on C++ [16]. One of the major advantages of Qt is the cross-platform compatibility. No matter the development operating system the produced application can be used across many different desktops, embedded and mobile platform. Most noteworthy is the usage across both Windows, OS X and Linux. Qt is based on the usage of different types of widgets such as buttons and text boxes. The widgets are added to the application and commanded using signals, layouts are used to control the position and distribution of widgets within the application. Third parties also allow free extensions allowing development in other languages than C++, which is utilized in this project as the Riverbank Computing extension PyQt5 [12] is used allowing development in Python.

QGraphicsScene

The user interface of the application is built up by a number of different widgets from Qt such as QTabWidget and QToolButton. All widgets are connected together in the MainWindow class. The most important component however is the QGraphicsScene as it is the component in which the creation and editing of the geometries is made. Using separate controls in the QGraphicsScene and overriding the native signals such as MousePressEvent and MouseMoveEvent the actions are controlled in the scene to produce the sought behavior while still utilizing many of the commands and objects native to the QGraphicsScene.

4.2 Core Functions

4.2.1 Converting Graphical Objects to a Gmsh Geometry Description

The program operates by performing the drawing and movement of geometries completely separate from the Gmsh code. When the editing is completed the Gmsh code is generated based on the existing geometry. Each added geometrical item is stored in a list. When the Gmsh code is generated, this list is looped through, and for each item the points and connecting edges are added. To generate a correct mesh some special cases are covered to ensure correct interaction between the geometrical items: overlapping points, point overlapping edge and overlapping edges.

Overlapping points

Overlapping points correspond to two geometrical items sharing a common node. It is therefore important that the Gmsh geometry is generated with this node as shared between the two items. Otherwise, two nodes having the same coordinates would be independent from one another, hence, the items would not work as a continuous unit. To handle this, a simple check is made whether a point already has been added in the Gmsh code on the same coordinates, if so, no new point is added, instead the already existing one is used.

Overlapping point on edge

By the same reasoning as for the overlapping points it is important that a corner point of one item that overlaps an edge of another item must be connected to that edge to ensure that the changes to the node affects both items. To handle this a check is passed for all items before the generation of the Gmsh geometry to determine if there is any point overlapping on any of the edges. If that is the case, then an extra point is added to the item in which the edge is overlapped. The point added essentially splits the edge in to two separate edges. Since there are now two points in the exact same position it is handled by the overlapping points explained earlier.

Overlapping edges

Overlapping edges consists of two scenarios: either the whole edge is overlapping (the items share an edge) or part of the edge is overlapping. If the whole edge is overlapping then the edge points of both edges are also overlapping. During the geometry generation one of the edges will be added first, without the knowledge of an overlap. When the second edge is to be added the overlap is detected. Instead of adding the second edge, the already added one is used. This edge is used to connect to the rest of the geometry and thus treating overlapping edges as the same edge.

In the case of partially overlapping edges, the solution is the same as for an overlapping point on an edge. As one of the edges is the shorter one, the end points of this edge overlap the edge of the other item. In the position where the overlap occurs, two new points are added on the edge. After this this scenario the same procedure is used as if the whole edge is overlapping.

4.2.2 Snapping of Points to Points and Snapping of Points to Edges

The overlapping described in the previous section is based on that the two intersecting points share the exact coordinates. It is often difficult for users to place points pixel perfect on top of each other. Therefore, snapping of points to either other points or edges is required. The snapping is enabled when moving a corner point in the scene.

Snapping from point to point

When the selected point is moved in the scene it is compared to a stored list of all corner points in the scene. Using the NumPy norm command the euclidean distance to all other points is calculated. If any of the other points (does not compare to itself) is within the distance of 10 units the selected point coordinates are overridden by the coordinates of the nearby point.

Snapping from point to edge

The snapping from a point to edge works slightly different. Here it is not feasible to store every single point of an edge in a list for comparison. Instead, every edge is stored and when a selected point is moved in the scene a square area of 10x10 pixels around the pointer is used. Every point in the area is checked whether it is contained in any of the edges. In that case the point is added to a temporary list. If there are multiple points, which mostly is the case since the square area likely overlaps more then one pixel of the edge, the temporary list is looped through and the point with the smallest euclidean distance to the pointer is selected as the point to snap to.

Worth noting is that an alternative to use grid snapping also exists. The point is then snapped to the closest point in the grid. However, it is possible to first create geometries with the grid snapping turned off. If the grid snapping is then turned on, the points created earlier are then possibly not on the grid. If the grid snapping would have priority, these points would then be unreachable while the grid snapping is turned on. Therefore, to ensure correct overlaps, it is prioritized so that the snapping to another point or edge has priority over snapping to the grid.

4.2.3 Views and Modeling Modes

The program consists of four different views which are controlled by a tab widget to ensure that only one of them is activated at a time. The pane of the tab bar also ensures that only the relevant tools are available for the activated view.

Surface View

The surface view is the main view of the program. This view is used to create, load, edit and move geometries. It also allows splitting of edges by adding an extra node.

Border View

The border view is used to display only the borders of the geometries and allow editing the markers of the points and edges.

Geometry View

The geometry view uses the CALFEM vis method of displaying the geometry. This allows the user to view the CALFEM geometry in the program to ensure that everything is working as intended.

Mesh View

Similarly as the geometry view the mesh view, displays the mesh using CALFEM vis. The mesh is generated using standard values, but controls allows the user to re-generate the mesh using other settings such as max element size and degrees of freedom per node.

4.2.4 Export/Import Files

In Python

The export and import of files are implemented in a separate Python module as a series of methods to allow it to be used independently of the program. In order to perform this, the Python library Pickle is used to save data to local files. Utilizing this tool the export command writes the target attributes to file in a specific order and the import command then loads them in the corresponding order. The export and import commands were made in three different versions. Each version to handle a stage at which the user could wish to save their work. To avoid issues with users loading invalid input, the three versions were each assigned a file extension:

- Save CALFEM geometry instance - .cfg
- Save CALFEM mesh instance - .cfm
- Save mesh arrays - .cfma

In the program only CALFEM geometry objects (.cfg) are allowed to be imported as the conversion needs to be made back to QtGraphics objects. This was relatively simple for geometry objects when only considering polygons as the structure is similar with nodes and connecting lines. Files saved only as a CALFEM mesh object (.cfm) or as the mesh arrays (.cfma) have a different structure that is too complex to convert back to a description of the geometry.

In MATLAB

To export the results to MATLAB a slightly different approach was taken. Here only the arrays from the mesh generation are relevant as the MATLAB version of CALFEM is not implemented based on Gmsh. To export the arrays to be loaded in MATLAB, the library Savemat from SciPy [15] was used. This tool converts NumPy arrays to MATLAB Arrays and Python dictionary to MATLAB struct and saves it in the native MATLAB .mat format. This produces a file that can be loaded into MATLAB. It should be noted that MATLAB does not accept numbers in variable names, meaning that provided boundary markers need to be provided as pure text in order to appear in the export.

4.3 Graphical User Interface

In order to implement a program that is easy to use and intuitive for new users the implementation was based on the idea of direct manipulation [17]. Direct manipulation is an idea of providing the user with constant visual impressions in direct response to performing actions in the program. When drawing geometries this is done by showing supportive nodes and lines during the creation which are updated with movement of the mouse. Once the surface is completed these supportive items are deleted and the actual surface is drawn.

The process of creating the geometries follows a relatively linear process, where the user creates a geometry, modifies the borders, generates a CALFEM geometry and if the geometry is satisfactory generate a CALFEM mesh. However, it is not a set order and the user should be allowed to jump back and forth between the different modes. Furthermore, only the tools that are relevant for the selected view should be available to the user. To combine these two requirements into the program a tab bar was used to handle the different modes where clicking a tab automatically switches mode. Each tab also has its respective pane, in this pane the different tools available are displayed and hence only showing the tools relevant for the current view.

In the pane it was determined to use a ribbon-like design, loosely based on the Windows guidelines [18]. A ribbon is a command bar that replaces traditional menu- and toolbars. It focuses on collecting similar tools in relevant groups and is intended to help users to find features and functions more easily. One reason to use a ribbon instead of a classical toolbar is that the ribbon allows for more space and text as well as other components such as a spin box. When using a toolbar some of the icons such as "draw rectangle" and "draw rectangle hole" may look very similar and complementing with text labels provides extra clarity to avoid confusion when using the program. The downside is that the ribbon takes up more space than a compact toolbar, however, since the program will always need a relatively large window size due to the graphics scene, the size of the ribbon is not considered an obstacle as it is relatively small in comparison.

4.4 Editor Code Interaction

The implementation comes with three different available functions used to launch the editor.

1. run_editor(): Initiates the editor as a stand-alone program without any input or output
2. run_editor_and_load(g): Initiates the editor with a CALFEM geometry object (g) as input, returns no output
3. g, marker_dict = cfe.edit_geometry(g): Initiates the editor with the option of providing a CALFEM geometry object (g) as input. On exit the editor returns a CALFEM geometry object (g) from the, at the time of exit, geometry in the editors graphics scene. It also returns a marker dictionary (marker_dict) which holds the marker names connected to marker indices in the geometry object.

Using option 1 or 2 the user has to manually save the work using the functions available in the user interface. These options are intended be used to launch the editor outside of a script . Option 1 when creating geometries to be used in future work which then are saved as local files. Option 2 similarly is used when there is a pre-existing CALFEM geometry object that the user wishes to edit, the modified geometry is then saved as a local file. Option 3 is the command used when calling the editor from within a script, as this will return the geometry description in the editor directly back in to the script. Hence this command is used when making minor changes that are returned directly back to the script. Although, since the only difference between the options is in how the program is launched all tools and functions are available in all cases. This way, using option 3 allows saving to local files as well as returning to the script.

5 Discussion and Conclusions

In this chapter some of the decisions made during the course of the project are described and motivated. Conclusions regarding the result of the work are discussed as well as future possibilities of development.

5.1 In-Work Decisions

In the initial stages of the development of the program there were two considered options of how to handle the geometrical shapes, using either geometrical shapes as surfaces or using nodes and different types of connections to combine.

5.1.1 Surfaces and Geometrical Shapes vs Nodes and Connections

The idea of drawing using surfaces is to use different simple geometrical shapes in combination with overlapping and other tools being able to create close to arbitrary geometries. The upside of this approach is that it is the most conventional way of drawing in other types of user friendly software such as PowerPoint and MS Paint. It also displays the geometry in a very intuitive way. It allows for simple editing if the user wants to change the geometry with slight modifications such as moving a hole to see the impact further in the process.

Using nodes to draw provides easier means of implementation as it is more similar to the structure of Gmsh where all geometries are defined by points with some sort of line connection. The downside is that the user has to know where to place all intersecting points and borders which is not always the most intuitive way of thinking about geometries. This may also cause some problems when wanting to move only a part of a geometry.

Furthermore, drawing with automatic connections drastically reduces the amount of actions needed to create a geometry, for a polygon drawing using surfaces requires n + 1 actions while using nodes and connections requires 3n actions, where n is the number of sides in the polygon.

From these two approaches it was decided to implement the drawing using surfaces and geometrical shapes. Even though this alternative was considered the more challenging, it was assumed to be the more user-friendly as a finished program.

5.1.2 Drawing Using Only Polygons

The initial idea was to be able to draw using four standard geometrical shapes: circle, rectangle, ellipse and polygon. Noteworthy is that circle is a special case of the ellipse and rectangle is a special case of polygon, which means it comes down to drawing using straight lines in the polygon and arced lines for the ellipse. After deciding on using the drawing based on surfaces and geometrical shapes it eventually became evident that there was a need to handle overlaps in the geometry. For single points the check of overlaps was no problem since methods for this exists in PyQt5. However, there was no easy way of checking the overlap of whole graphical items. To handle this, manual methods had to be manufactured to exhibit the desired behavior. This posed a challenge for both polygons and ellipses. However, the methods could be constructed with significantly less complexity utilizing that polygons only have straight lines, compared to the arced lines of an ellipse. This is due to the fact that knowing all the points automatically translates to knowing all the lines since two points are always connected with the same straight lines. The problem with the arced lines is that knowing the points does not provide any information on the arc of the curve and is hence ambiguous. One example of this is, if two points are known to be contained within a square. If the line is straight it is equivalent to that the line is contained within the square as seen in Figure 5.1. However, this is not the case for the arced line as the curvature of the line can cause it to intersect the edge of the square, even if both end points are inside the square. It was therefore decided during the project to focus solely on implementing the structure for the polygons and leave the ellipses for future work.

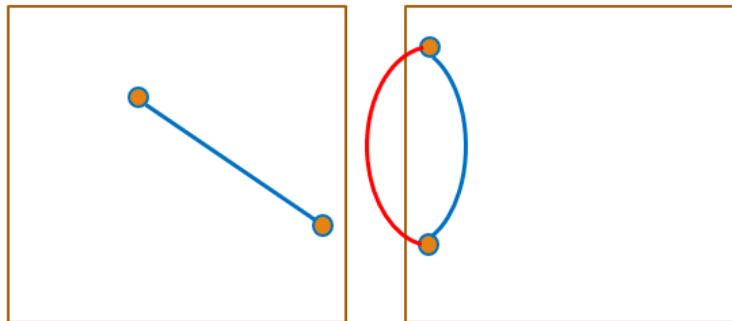


Figure 5.1: Comparison of how a straight line is contained within a square where the end points are contained while arced line is contained depending on the curvature of the line.

5.2 Conclusions

The aim of the project was to implement a program with the possibility to create and edit 2D geometries. This aim has been achieved as a program has been developed

and is operational. However, there were further goals set regarding the functionality surrounding the program. The demand on the program to be easy and intuitive to use is of course not objective, as the idea about this varies from person to person. The idea was to conduct a user study to validate whether the design was considered to be intuitive or not. Unfortunately, due to the time frame, the program was not finished in time to enable a user study that would have provided useful results. Regarding the programs versatility, the goals were met in regards to being able to launch the program as a stand-alone program and from command with option of returning a geometry, also with loading pre-existing geometries. To use as a component in other Qt-based graphical user interfaces, the goal was partially met as the graphics scene holds all the core functions. It can be used without the main window allowing it to be used as a resource with a separate user interface. However some extra functionalities are based on the existence on the controls and widgets available in the main window user interface.

5.3 Future Work

For future work there exists many possibilities to improve and further elaborate on the implemented functions as well as introducing further functionalities.

5.3.1 Improvement on Existing Functions

In the existing functionality one of the major drawbacks is the poor computational efficiency of the snapping to edges. When a point is moved, all the edges are looped through and thus it becomes increasingly inefficient the more edges are added which could cause issues if creating a large or complicated geometry. To solve this, more thought could be put into how to do this more efficient and perhaps make use of matrix operations such as for the point to point snapping which is much more efficient. Another option would be to explore if any other data structure could be used with more efficiency.

Regarding the absence of a user study, the program might need some improvements in regards to usability to be more intuitive to use. Further, it would be very useful to find out what could use improvement or need alteration.

5.3.2 Extended Functionality

Missing from the early stages of the work is the possibility of creating other shapes than polygons, more specifically being able to create ellipses and circles. Due to the focus on only creating surfaces, the usage of creating lines was also excluded. Adding this would however require the user to manually define which lines should build up a surface which introduces a second way of creating surfaces. This would combine the idea of drawing using surfaces or nodes and connections, allowing the user to do the same action with different commands, which was considered as contradicting to

the idea of keeping the program simple and intuitive. Another option would be to implement automatic closure when the lines form a surface, but this would likely be very complex to implement and be more prone to mistakes of the user as all lines must form a closure in order to generate a mesh.

Another useful tool that could be implemented to be able to create more arbitrary geometries in a more efficient way is a cut out function. This would work similar to the already existing merge ability where two polygons are combined to one, with the difference that one polygon would be chosen as the one cut out in the other. I.e. the area of the cut out polygon and the overlap with the existing polygon would be removed, thus cutting out a part of the previous polygon. Another way of implementing a similar functionality would be to modify the current hole creating function to allow overlap on edges; if the hole then removes the overlapping edge it would behave similar to a cut out as one or more edges of the hole would not be contained within the polygon.

5.3.3 Computational Efficiency

The most notable problem with computational efficiency comes with the snapping to edges algorithm. Snapping to points could be made very efficient as it was possible to apply matrix operations with the coordinates. However, with the edges it is not feasible to check every point on the edges as there exists a vast amount. Instead the iterations loop through every edge to check if it is in the vicinity of the mouse pointer, which is not as efficient. As this has to be done with every pixel movement the increased computational time becomes noticeable and shows as a slight lag when moving the pointer and does not appear as smooth. The problem increases with larger amount of edges in the scene. Yet, since the geometries created are not expected to be of very high complexity the number of edges should remain relatively low to the degree that the lag should not create any major issues.

Bibliography

[1] *CALFEM for Python*. https://github.com/CALFEM/calfem-python. Accessed: 2021-09-07.

[2] *CALFEM for MATLAB*. https://github.com/CALFEM/calfem-matlab. Accessed: 2021-09-07.

[3] Matti Ristinmaa, Göran Sandberg and Karl-Gunnar Olsson. "CALFEM as a Tool for Teaching University Mechanics". In: *CAL-laborate International* 5 (Aug. 2000).

[4] *MATLAB*. https://se.mathworks.com/products/matlab.html. Accessed: 2021-09-07.

[5] Andreas Ottosson. "Implementation of CALFEM for Python". MA thesis. Division of Structural Mechanics, Lund University, 2010.

[6] *Python version 3.8.11*. https://docs.python.org/3.8/faq/general.html. Accessed: 2021-09-07.

[7] *What is NumPy*. https://numpy.org/doc/stable/user/whatisnumpy.html. Accessed: 2021-08-23.

[8] Andreas Edholm. "Meshing and Visualisation Routines in the Python Version of CALFEM". MA thesis. Division of Structural Mechanics, Lund University, 2013.

[9] *Gmsh: A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*. https://gmsh.info/. Accessed: 2021-08-02.

[10] *Partial Differential Equations Toolbox*. http://matrix.etseq.urv.es/manuals/matlab/toolbox/pde/pdetool.html. Accessed: 2021-08-04.

[11] *QGraphicsScene Class*. https://doc.qt.io/qt-5/qgraphicsscene.htmll. Accessed: 2021-09-07.

[12] *What is PyQt?* https://riverbankcomputing.com/software/pyqt/intro. Accessed: 2021-08-02.

[13] *Microsoft PowerPoint*. https://www.microsoft.com/microsoft-365/powerpoint. Accessed: 2021-08-23.

[14] *pickle — Python object serialization*. https://docs.python.org/3/library/pickle.html. Accessed: 2021-08-23.

[15] *scipy.io.savemat*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.savemat.html. Accessed: 2021-08-23.

[16] *About Qt*. https://wiki.qt.io/About_Qt. Accessed: 2021-08-02.

[17] D. Åkesson and Caitlin Mueller. "Using 3D direct manipulation for real-time structural design exploration". In: *Computer-Aided Design and Applications* 15.1 (Jan. 2018). ISSN: 1686-4360. DOI: 10.1080/16864360.2017.1355087.

[18] *Ribbons.* https://docs.microsoft.com/en-us/windows/win32/uxguide/cmd-ribbons. Accessed: 2021-08-11.

Appendix A

Manual

CALFEM Geometry Editor Manual

September 12, 2021

Manual describing the contents and controls of the CALFEM Geometry Editor. Future revisions will be available from: `https://calfem-for-python.readthedocs.io/en/latest/`

Contents

1 Running the Editor **2**

1.1 run_editor 2

1.2 run_editor_and_load 2

1.3 edit_geometry 2

2 Interface Description **3**

2.1 Interface Overview 3

2.2 GraphicsScene Controls 4

2.3 Grid Controls 5

2.4 Load Geometry 6

2.5 Geometry 7

2.6 Edit Polygon 8

2.7 Border Controls 9

2.8 CALFEM geometry 10

2.9 CALFEM Mesh 12

1 Running the Editor

There are three ways of launching the editor, which one chosen depends on if there is any input and if any output is desired

1.1 run_editor

Description

Launches the CALFEM Geometry Editor as a stand alone program

Syntax

cfe.run_editor()

1.2 run_editor_and_load

Description

Launches the CALFEM Geometry Editor with an input geometry g, geometry needs to be of object type CALFEM.geometry.Geometry

Syntax

cfe.run_editor_and_load(g)

1.3 edit_geometry

Description

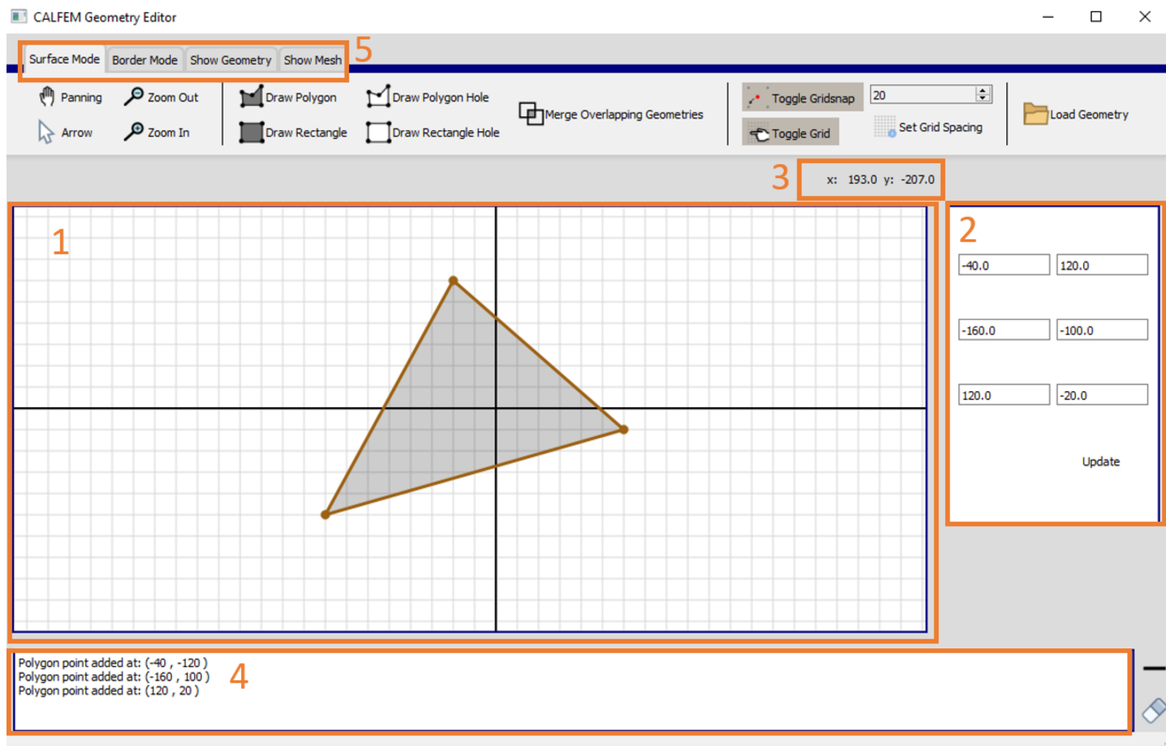
Launches the CALFEM Geometry Editor with an input geometry g, geometry needs to be of object type CALFEM.geometry.Geometry. Returns output geometry and dictionary of marker names

Syntax

g, marker_dict = cfe.edit_geometry(g)

2 Interface Description

2.1 Interface Overview



The user interface is mainly built up by 5 different parts:

1. The GraphicsScene which is the canvas used to do all the drawing and editing of geometries.

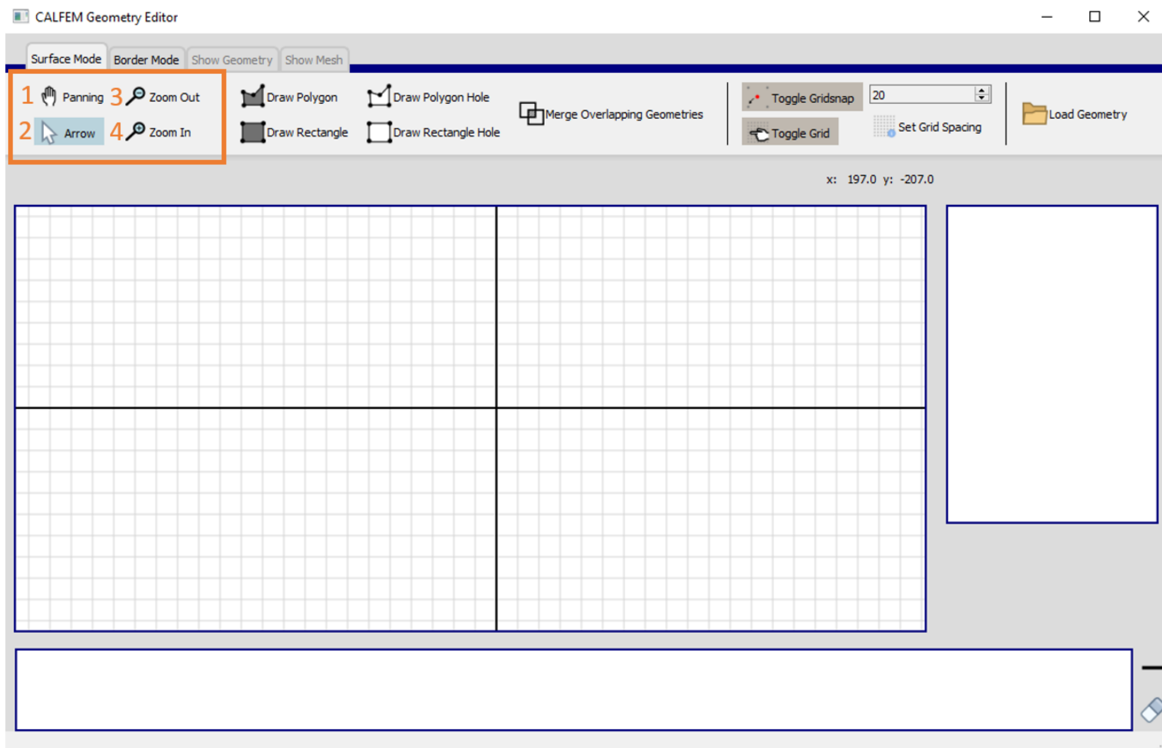
2. Scroll area, extra display area used to show certain controls when needed

3. Coordinate labels showing the current coordinates of mouse cursor in relation to the center of the GraphicsScene

4. Console, printing the history of commands and actions used

5. Ribbon toolbar, holds the controls for all the tools and options, changing between the tabs both changes the viewing mode but also changes the available tools to only display the relevant ones for the current view.

2.2 GraphicsScene Controls



1. The panning button is pressed to toggle the panning state, during this pressing the mouse button while in the GraphicsScene and moving the mouse will result in the GraphicsScene changing the displayed subsection of the canvas

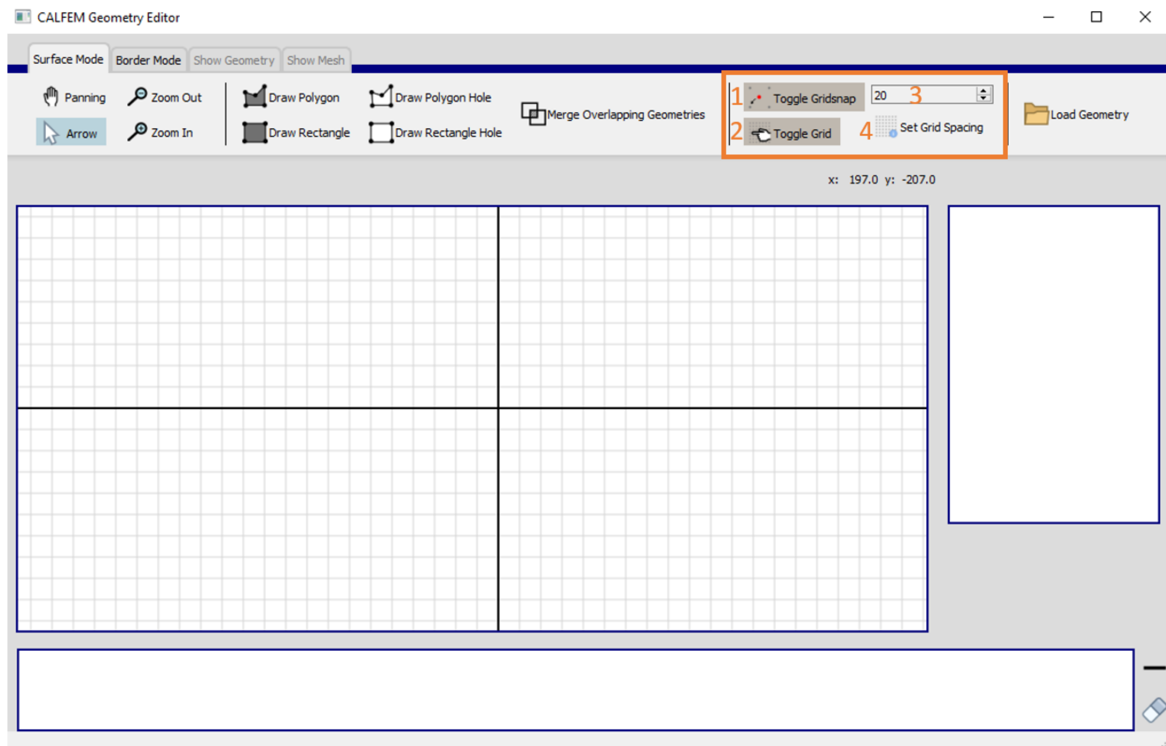
2. The arrow button is pressed to toggle the arrow mode, this is the main mode of the GraphicsScene and controls all events not handled by any other button. This includes moving geometries and dragging corners.

3. Pressing zoom in will increase the scale of the GraphicsScene with a factor 2

4. Pressing zoom out will reduce the scale of the GraphicsScene with a factor 2

Note that both zoom in and zoom out may also be controlled by the scroll wheel

2.3 Grid Controls

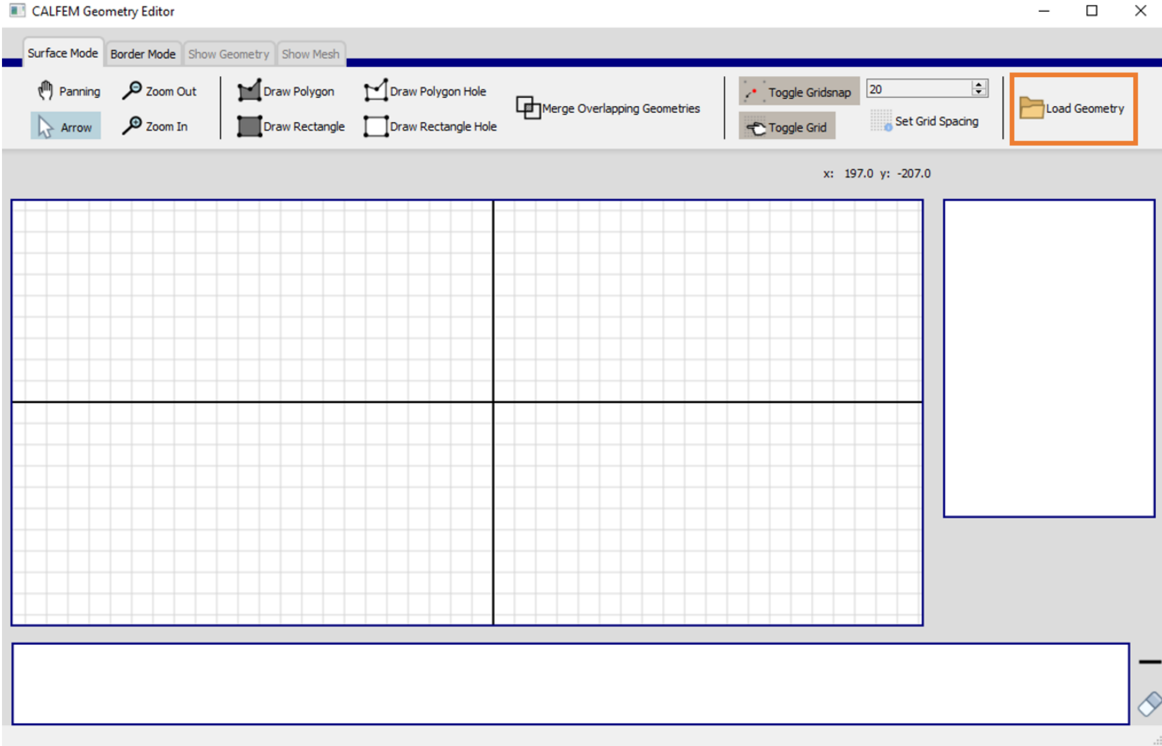


1. Toggle Gridsnap - Toggles the activation of snapping to the grid. When activated drawing and moving surfaces and nodes will automatically snap to the closest grid intersection point. Note that the snapping can be enabled even without showing the grid.

2. Toggle Grid - Toggles the display of the grid lines

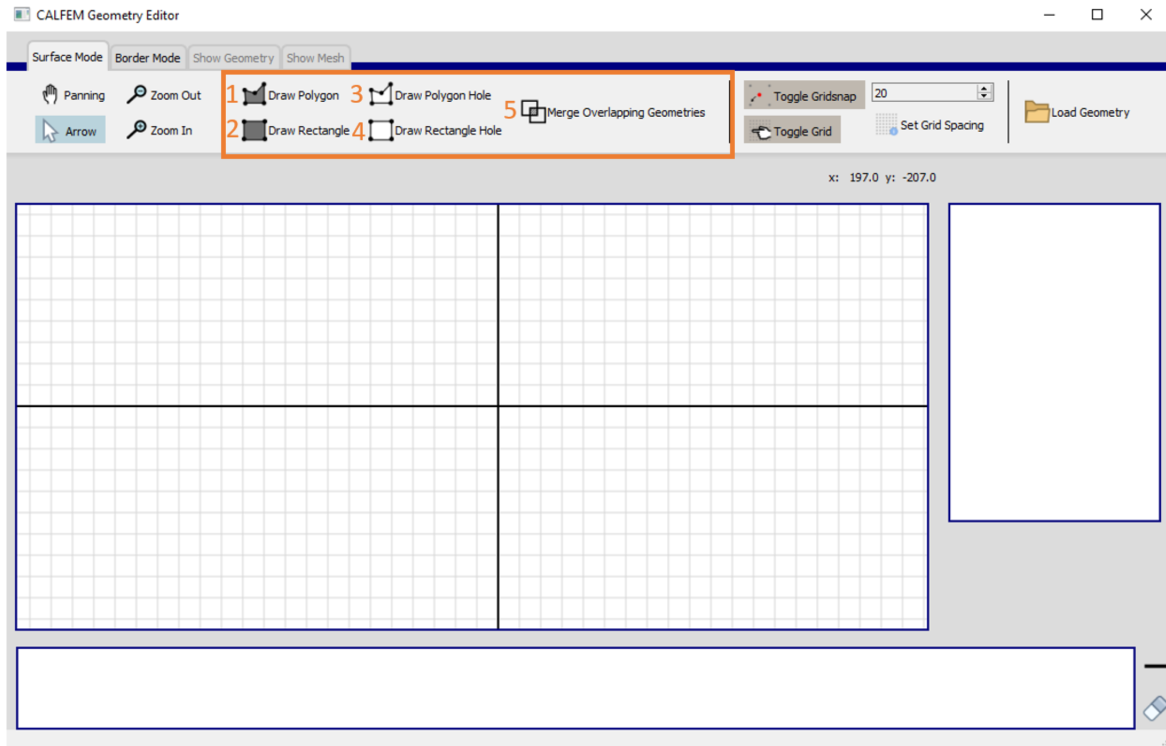
3. + 4. Combined spinbox and button, the spinbox displays the grid spacing (the distance between the lines in the grid), this can be edited but to take effect in the GraphicsScene the Set Grid Spacing button must be pressed.

2.4 Load Geometry



Tool to load a pre-existing geometry into the program, when pressing the Load Geometry button a dialog window will be opened. Here a .cfg file can be chosen which will be loaded in to the GraphicsScene.

2.5 Geometry



1. Draw Polygon - Toggles create polygon mode, clicking the left mouse button in the scene will create a node in the polygon. The nodes are automatically connected in a successive order, pressing the right mouse button will connect the most recently drawn node to the first node and hence close the polygon.

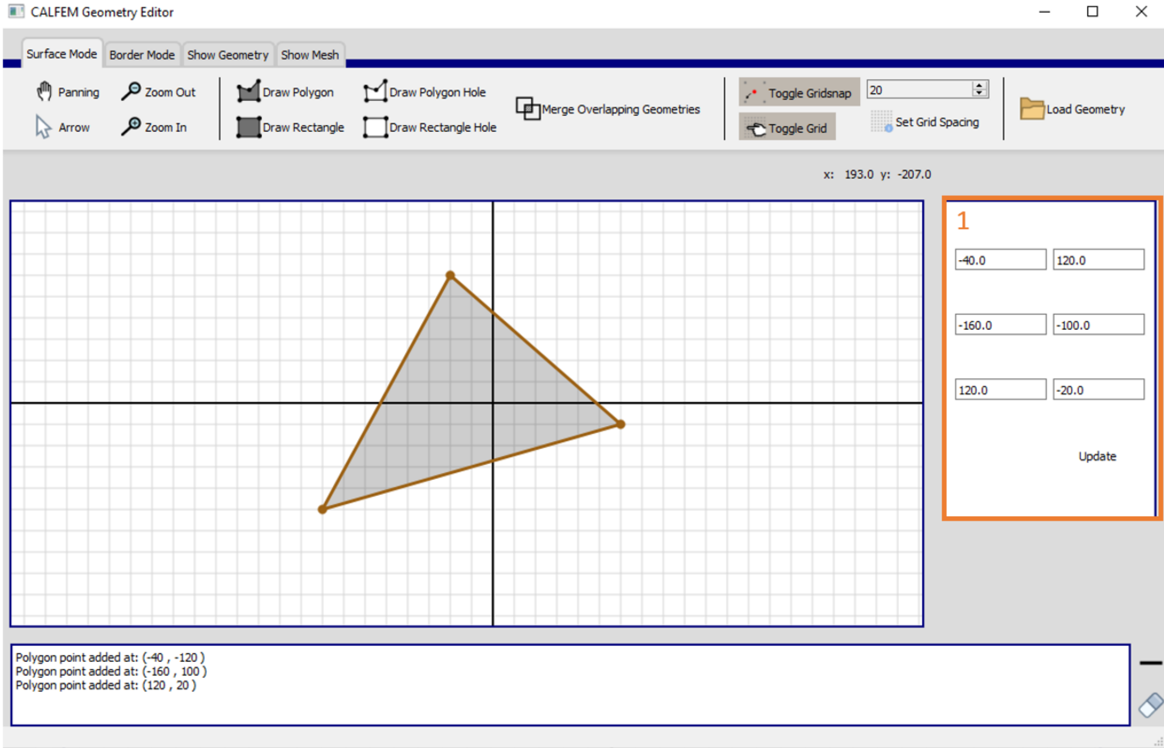
2. Draw Rectangle - Toggles the create rectangle mode. Clicking will create the starting point of the rectangle, when moving the mouse will show a supportive rectangle with the current mouse position as the second corner point, clicking again will create the real rectangle at the current position.

3. Draw Polygon Hole - Toggles the create polygon hole mode, works the same as creating a polygon but appears with a white background and acts as a hole in later steps and must therefore be completely enclosed within a outer polygon or rectangle

4. Draw Rectangle Hole - Toggles the create rectangle hole mode, works the same as creating a rectangle but appears with a white background and acts as a hole in later steps and must therefore be completely enclosed within a outer polygon or rectangle

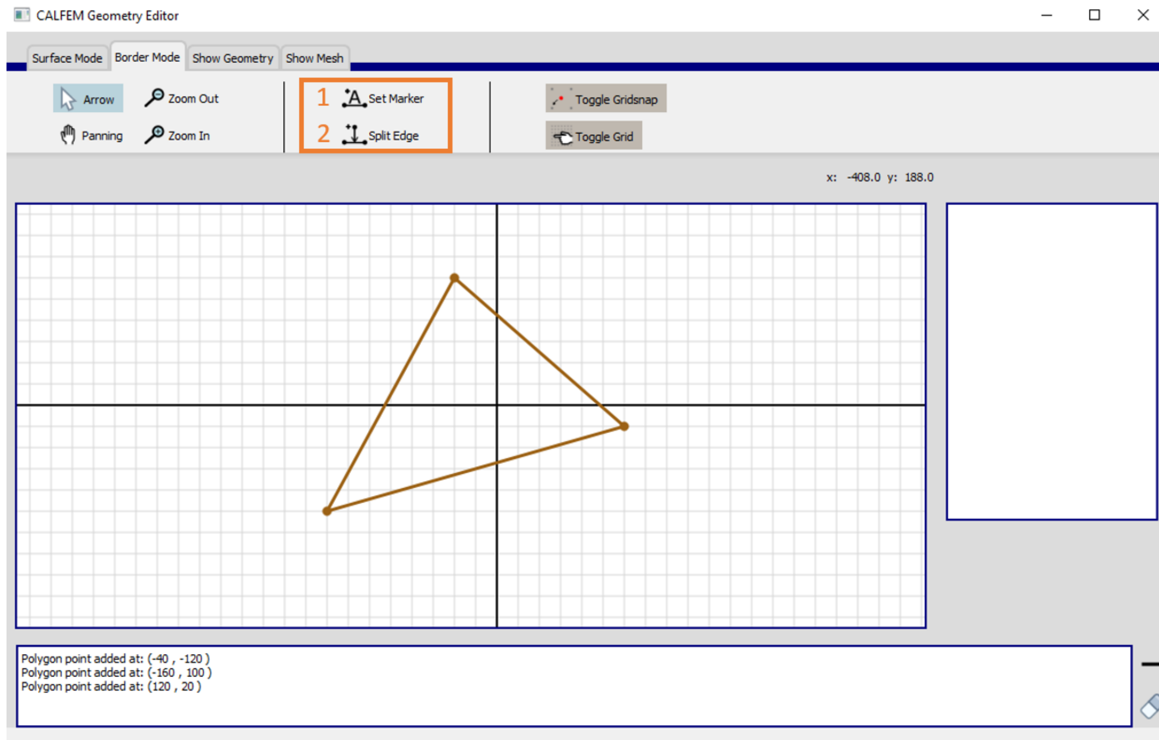
5. Merge Overlapping geometries - When pressing this geometries which overlap each other (excluding completely contained geometries and holes) will be merged together forming one new polygon with the boundaries of the two overlapping geometries.

2.6 Edit Polygon



The edit polygon dialog is shown upon double-clicking on a polygon surface. This will show all the coordinates of the polygon and allow editing of these. Pressing the Update button will update the edited position of the polygon points.

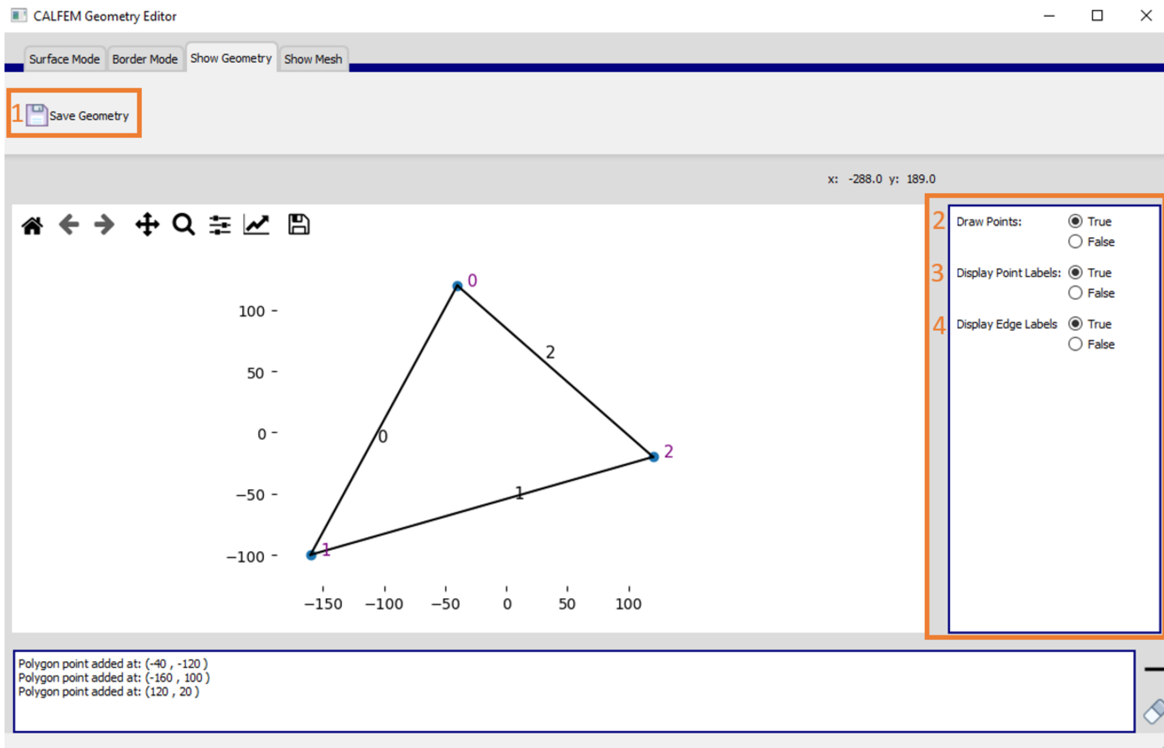
2.7 Border Controls



1. Set Marker - Toggles the set marker mode, to set a marker press a point or edge. The pressed item will be highlighted and the pane on the right side will display a textbox in which the wished marker name can be set. Note that MATLAB only accept markers pure text i.e. no numbers or special signs. Press the "Set Marker" button to update the edge or point with the new marker, the target point or edge will then turn red and the marker text appear.

2. Split edge - Toggles the split edge mode, moving the cursor close to the target edge will cause the split edge node to appear in gray. When at the wished position press the mouse to add the new node.

2.8 CALFEM geometry



The CALFEM geometry is automatically created with whatever is produced in the surface/border view and shown in the GraphicsScene when toggling to the view using the tabs.

Error messages

Toggling with incorrectly drawn geometries may produce error messages to avoid issues with CALFEM.

- Overlapping geometries: Occurs when either two surfaces intersect each other than only at the border, also occurs when a hole surface is not fully contained or on the border of the outer surface

Saving CALFEM geometry

Pressing the only available tool "Save Geometry" in the toolbar prompts the file dialog to select a target folder. The file is saved with a .cfg file extension. To load the geometry object in another script use the load_geometry command from CALFEM.utils

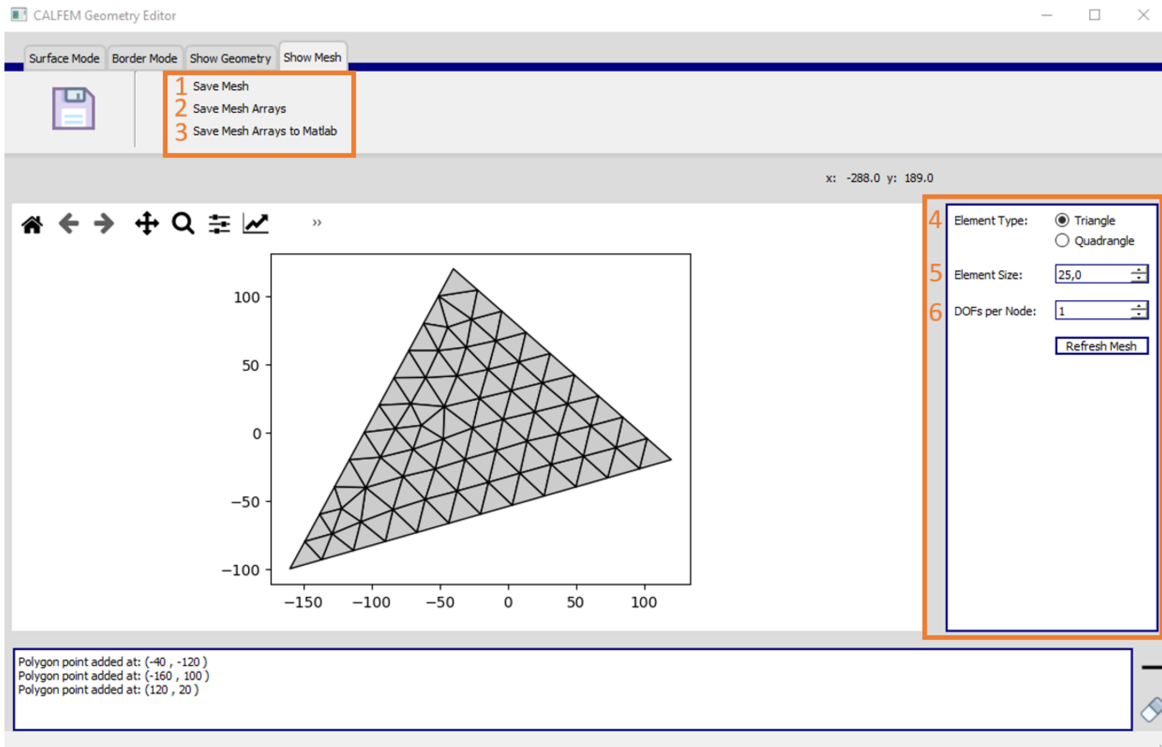
Display Geometry Controls

When displaying the geometry there are some display options available, pressing any automatically refreshes the geometry with the new options:

2. Toggle whether to display the corner points

3. Toggle whether to display the index of each corresponding corner point, note that disabling the corner points automatically also disables this option
4. Toggle whether to display the index of each edge along the border

2.9 CALFEM Mesh



The CALFEM mesh is automatically created with whatever is produced in the surface/border view and shown in the GraphicsScene when toggling to the view using the tabs.

Error messages

Toggling with incorrectly drawn geometries may produce error messages to avoid issues with mesh generation.

- Overlapping geometries: Occurs when either two surfaces intersect each other than only at the border, also occurs when a hole surface is not fully contained or on the border of the outer surface

Saving CALFEM mesh

In the ribbon toolbar there are three only three available commands which are the different options to save the mesh, each prompts the file dialog to select a target folder:

1. Save Mesh - saves the mesh as a file with .cfm extension, to load the mesh object in another script use the save_mesh command from CALFEM.utils

2. Save Mesh Arrays - saves the mesh as a series of arrays and dictionaries (the same as when using cfv.create), to load the arrays in another script use the save_arrays command from CALFEM.utils

3. Save Mesh Arrays to MATLAB - saves the mesh as a series of MATLAB arrays and cells, the file is saved in the standard MATLAB .mat format and loaded in MATLAB using "load()"

Display Mesh Controls

When displaying the mesh there are some mesh settings available, after editing settings press "Refresh Mesh" to apply the settings:

4. Element type - As of now only the only two available element types Triangle and Quadrangle are available from CALFEM
5. Element size - The factor of which the element sizes are multiplied
6. DOFs per Element - Number of degrees of freedom per node, integer value