Documentation for Ultimaker Cura Plugin “FitnSit”

Developed as Part of the Software Practical Course “Multimodal Media Madness” by Michael Assad and Karine Louis

1. Introduction

The UM Cura plugin “FitnSit” was developed as part of the Software Practical Course “Multimodal Media Madness” at the RWTH Aachen University in Summer 2020. FitnSit aims to facilitate and automate cutting oversized 3D STL-Models into smaller ones. Users of the plugin are not expected to possess any prior knowledge of or experience with CAD-Software.

2. Software

The frontend Code of FitnSit was written in PyQt5, mainly utilizing the QtQml Module. The backend code was written in pure Python.

2.1. Frontend

2.1.1. Dependencies

The plugin has a dependency on QtQuick, the standard library for writing QML applications. The Qt Quick module provides all the basic types necessary for creating user interfaces with QML. It provides a visual canvas and includes types for creating and animating visual components, receiving user input, creating data models and views and delayed object instantiation.

The plugin also has a dependency on QtQuick QML types such as Qt Quick Controls, Qt Quick Dialogs and Qt Quick Window.

QtQuick.Window contains types for creating top-level windows and accessing screen information.

QtQuick.Dialogs contains types for creating and interacting with system dialogs.

QtQuick.Controls provides a set of controls that can be used to build complete interfaces in Qt Quick.

The plugin has a further dependency on Uranium (UM), which is a Python framework for building 3D printing related applications. Cura is built on top of the Uranium framework.

2.1.2. Main Frontend Components

Below we describe the main frontend components which are used to build the User Interface. Please refer to the thoroughly commented frontend code for further information about the below mentioned components.

A. UM Dialog

The base of the User Interface is a Uranium dialog, titled as “FitnSit”. It has the main functionalities of the QtQuick dialog, is however adjusted to the theme of Ultimaker Cura. The dialog is a popup used for the interaction with the user. The dialog uses a **Tab View** control to provide tab-based navigation for the three main plugin functionalities: Automatic Cutting, Graphical Cutting and Advanced Cutting. The Dialog furthermore includes a **ComboBox** which is used to display the results of the cut, a “Deploy” **Button** used to deploy the selected Mesh in the ComboBox onto the Cura workspace, a “View” **Button** used to view the selected Mesh in the ComboBox in a default 3D-Viewer and **Text** messages to display whether a selected cut will result in a watertight object. The default visibility of the ComboBox, the Buttons and the Text boxes is set to false.

B. Automatic Cutting Tab

The Automatic Cutting tab uses a **SpinBox** for a user input of the number of parts the object should be cut in. The SpinBox is capped at a maximum Value of 10 so the cut is limited to 10 parts. Furthermore, the tab uses a set of three **RadioButtons** for the three axes: x, y and z. The RadioButtons are grouped exclusively allowing the user to choose only one of the three axes. They are highlighted using the same color coordination of the Cura axes. To allow capping the object the tab includes a **CheckBox.** When clicking one of the RadioButtons or the CheckBox, the backend function *automaticCutRoutine()* is called with the SpinBox value, the selected Axis, the Boolean value of whether the CheckBox is checked and the tester set on true. This tests whether the selected cut will result into a watertight object. If the tester function was successfully executed a message saying the selected cut results in a watertight object is shown. Elsewise a warning message saying the selected cut results in a cut that is not watertight is shown. To execute a real cut, the tab includes a normal **Button** positioned at the bottom right of the Tab, labeled as “Cut”. Upon clicking the Button, the backend function *automaticCutRoutine()* is called with its respective parameters and tester set on false. After executing the cut, a frontend developed function **showResults()** is called, which is described in the frontend functions section below.

C. Graphical Cutting Tab

The Graphical Cutting tab uses four **RadioButtons** for the four views: Front, Top, Left and Right View. The RadioButtons are also grouped exclusively so the user can only choose one view to cut in. When clicking one of the four views, the frontend function **resetImage()** is called, which is also

described in the frontend functions section below. Afterwards one of the frontend functions **setRecSizeFront(), setRecSizeTop()**, **setRecSizeLeft()**, or **setRecSizeRight()**, depending on which view is clicked. Because the function resetImage() delivers images that are not correctly rotated, the images are rotated correctly when clicking one of the RadioButtons. To allow capping the object the tab includes a **CheckBox**, analogous to the Automatic Cutting CheckBox. Different than the Automatic Cutting the Graphical Cutting includes a **Rectangle**, highlighted with blue borders, to display the outline of the selected object in the Cura workspace. The Rectangle uses 5 nested images, each for its respective slice, to depict this outline. To enable the drawing of the desired cutting line on the image, we use a **Canvas** and a **MouseArea** for user input, hence the name "Graphical" cutting. The cutting line is drawn using the onPressed and onReleased mouse signals of the MouseArea. Upon receiving the signal onReleased, the backend function *manualCutRoutine()* is called with the coordinates of the mouse press and mouse release, the selected view, the boolean value of the checkbox and tester set to true. Analogous to the automatic cutting, this functionality allows testing whether the selected cut will result in a watertight object or not. To execute a real cut, the tab also includes a normal **Button** positioned on the bottom right of the Tab, labeled as "Cut". Upon clicking the Button, the backend function *manualCutRoutine(*) is called with its respective parameters and tester set on false. It is worth noting here, that these parameters are normalized to correspond to their real positions on the original object. To do that we use the backend functionalities *getMeshInfoX()*, *getMeshInfoY()* and *getMeshInfoZ()*. After executing the cut, the same frontend developed function **showResults()** is called, which is described in the frontend functions section below.

D. Advanced Cutting Tab

The Advanced Cutting Tab includes four **SpinBoxes** to enable cutting the object along a coordinate specified line. Similar to the Graphical Cutting Tab, this tab also uses four **RadioButtons** for the four views: Front, Top, Left and Right View. The RadioButtons are also grouped exclusively so the user can only choose one view to cut in. When clicking one of the four views the selected cut is tested for watertightness using the backend function *ManualCutRoutine()*, which is called with the spinbox values, the selected View, the Boolean value of the CheckBox and tester set on true. Analogue to the Automatic Cutting the advanced tab uses a **CheckBox** to enable capping the object and testing whether the selected cut will result in a watertight object or not. To execute the real cut, the tab also includes a normal **Button** positioned on the bottom right of the Tab, labeled as "Cut". Upon clicking the Button, the backend function *manualCutRoutine(*) is called with the SpinBox values, the selected view, the Boolean value of the CheckBox and tester set on false. After the cut function is called, the same frontend function **showResults()** is called. Different than the other two tabs, this tab uses a "Grab Dimensions" **Button** which calls the backend function *manager.getMeshInfo(1)*.

2.1.3. Main Frontend Developed Functions

A. showResults(i)

The showResults(i) function is called with parameter returnFlag. The returnFlag is set to the return value of the backend functions *automaticCutRoutine()* and *manualCutRoutine()*. If the cut was successfully executed the function is called with returnFlag = 1 and if the cut was not successful the function is called with returnFlag = 0. When showResults is called from the Automatic Cutting Tab the function uses an extra frontend developed function **createArray(n)** which is called with the SpinBox value of the automatic cutting to creates an array with the amount of Meshes the user chose to cut its object in. If the returnFlag == 1, the ComboBox is appended with the Array and the visibilty of the ComboBox, Deploy Button and View Buttons set to true. If the returnFlag == 0, the ComboBox List is changed to [“No Meshes”] and disabled. The two Buttons are also disabled. Similarly when the function is called from the Graphical or Advanced Cutting Tab instead of creating an Array the ComboBox is set to [“Mesh1”, “Mesh2”].

B. resetImage()

This function is only called in the Graphical Cutting Tab and is used to call the backend function *grab2DSlice(selectedView).* Afterwards the source of the 5 slices is set to the generated slices of the object. Because *grab2Dslice()* returns not correctly rotated slices, we rotate the slices correctly. This function ensures that each time one of the RadioButtons is clicked, the correct image of the selected view is displayed.

C. setRecSizeFront()

SetRecSizeFront() is also only called in the Graphical Cutting Tab. The function is called when the “Front View” RadioButton is clicked to scale the height and width of the displayed image to the height and width of the original object in Front View. It uses the backend functions *getMeshInfoX()* and *getMeshInfoZ()* to determine the height and width of the real object.

D. setRecSizeTop()

Analogues to **setRecSizeFront()** but for “Top View”. Uses the backend functions *getMeshInfoX()* and *getMeshInfoY()* to determine the height and width of the real object.

E. setRecSizeLeft()

Analogues to **setRecSizeFront()** but for “Left View” and “Right View”. Uses the backend functions *getMeshInfoY()* and *getMeshInfoZ()* to determine the height and width of the real object.

2.2. Backend

All frontend queries are handled in the backend.

2.2.1. Dependencies

FitnSit has both hard and soft dependencies on a number of python packages.

2.2.1.1. Hard dependencies

The Plugin has a hard dependency on trimesh. Trimesh is a pure Python library that specializes in 3d image processing. To fully utilize the plugin, we recommend using a release version above 3.7.4. Trimesh is used in processing 3d-cuts. Trimesh has a hard dependency on numpy. FitnSit utilizes in this regard the package importlib to allow trimesh to be loaded and imported.

The plugin also has a dependency on the packages UM and cura to interact with the Cura software. Further dependencies on os and sys are used to facilitate path extraction to make the plugin cross-platform.

PyQt5 is used in making the backend-frontend integration possible.

Out of above-mentioned packages, only trimesh needs to be externally delivered with FitnSit. The rest is already part of Cura.

2.2.1.2. Soft dependencies

The plugin utilizes a multitude of Python packages as soft dependencies. Among others rtree, scipy and shapely. Both scipy and shapely are delivered with Cura. Rtree is externally delivered with the plugin within the trimesh package.

2.2.2. Logic of main backend functions

We describe below how the logic of the main backend functions is built. Please refer to the thoroughly commented and documented backend code for further information about the below mentioned functions. You may refer to the commented source code as well for information about other functions not mentioned in this documentation.

A. adjustView(pt, view)

adjustView is used to adjust a 2-coordinate-based point given as float[2] into 3-coordinates as float[3] by filling up the missing dimension with zeros.

Every 3d-object observed from a planar view, i.e. not isometric view, can be displayed as a 2d-image. Over the discourse of this plugin, we mainly work for simplification purposes with 2-coordinated-based points. In order to convert these 2d-points back to 3d, we fill up the third dimension with zeros. Whether the zeros are filled up in place of x-axis, y-axis or z-axis, is dependent on the given view.

B. normalPt(pt1, pt2):

normalPt is a function used to find a normal vector to the line drawn between two given points. This function is used to find the normal vector to a slicing plane.

Given two points on a 2d-plane, it is easy to find the equation of the line passing through them by means of calculating the derivative **y'**. The normal vector to this line has by definition a derivative of **-1/y'**. As in vector calculations, vectors pass by definition through the origin point. That spares the need to calculate the eqn. constant.

C. manualCut(mesh, pt1, pt2, view, cap)

FitnSit is mainly built around this function. manualCut receives the mesh that is to be processed upon as well as two 2d-planar points as float[2]; i.e. four float coordinates. It receives a string indicating the view these points are to be understood in. The function received a further flag "cap" as well, that sets whether the cut models are to be capped.

As indicated in the hard dependencies, FitnSit uses trimesh to process 3d-cuts. Trimesh offers the function **trimesh.Trimesh.slice_plane(plane_origin, plane_normal, cap)** that slices a mesh with a plane, returning a new mesh that is the portion of the original mesh to the positive normal side of the plane. **plane_normal** indicates a normal vector to the slicing plane. **plane_origin** (*(3,) float*) is a random point on the slicing plane.

As a prestep to calling **slice_plane**, **manualCut** generates **plane_origin** and **plane_normal** using the two given points along with the view by means of the above described logic in **normalPt** and **adjustView**.

As **slice_plane** returns a single mesh to the positive normal side of the cutting plane, we call **slice_plane** a second time with the element-wise additive inverse of **plane_normal** to obtain the opposite looking mesh to the first obtained one.

D. manualCutRoutine(pt1x, pt1y, pt2x, pt2y, view, cap, tester)

manualCutRoutine receives all needed parameters to call in return manualCut where the currently selected mesh is cut.

In addition to the parameter required by **manualCut**, **manualCutRoutine** receives an additional flag: tester. When set, manulCutRoutine returns either boolean True if the mesh cut at the given view and points would be watertight, otherwise it returns boolean False.

E. automaticCutRoutine(num, axis, cap, tester)

At its core, **automaticCutRoutine** calls **manualCut** successively over the currently selected mesh until it is cut into "num" equally-sized meshes.

For this, **automaticCutRoutine** has to calculate the maximum dimension "n" of the bounding box of the currently selected mesh according to the given axis. It then computes the line eqn. at n/num and incrementally calls **manualCut** on the negative portion of its previous return with two points computed to lie over the line n/num.

The flags "cap" and "tester" function as described above in **manualCutRoutine**.

This results in cutting the currently selected mesh into num-meshes that are equally sized as seen from the given axis. This is arguably the easiest way to use the plugin to split an oversized model into smaller ones that are easier to print.

2.2.3. Capping

Capping is pre-implemented in trimesh. It relies on an external C++-engine though to triangulate the polygons generated by the cutting surface. In order to avoid importing an additional C++-library into Cura, we altered the flow of the capping function to use **scipy.spatial.Delaunay**. Further information about Delaunay algorithm can be found at https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Delaunay.html and https://en.wikipedia.org/wiki/Delaunay_triangulation

- **Capping limitations**

The capping is limited only to watertight meshes. It's not possible to accurately calculate the polygons generated by the cutting surface at non-watertight meshes.

The results from a capped cut are not guaranteed to be watertight either, as possible float inaccuracies and fluctuations might render the resulting meshes non-watertight. This has a direct effect on **automaticCutRoutine** with num>2. If one of the successively cut meshes isn't watertight, the whole chain is broken and **automaticCutRoutine** fails. In such cases, an appropriate message is displayed indicating that **automaticCutRoutine** was not successful.

2.2.4. Face projection

The backend is able to generate at command five 2D-slices of a given model at equal distances through the model's depth as observed from a given view. These slices are saved as a .svg formatted vector file.

A. grab2DSlice(origView)

Given a specific view as str, this function generates five 2d-slices as described above at equal distances according to the depth of the object as observed from the parameter origView.

B. Face projection limitations

As this function only takes five 2d-slices and presents them on top of one another, it's possible that some details without enough depth at either ends of the object as observed from origView are lost. It is also possible that the constructed shape from laying these slices on top of one another may look distorted and different from the expected orthographic projection of the original object.

One possible way to improve this limitation is to grab even more 2D-slices at smaller distances from one another. This fix is not efficient though and caused higher loading times of the plugin. We decided to omit this potential fix and remain at the above described approach.

2.2.5. I/O

The frontend-backend communication in terms of .stl and .svg file-exchange mainly goes through the **//export** folder.

This folder is emptied every time Cura is reloaded. It is also emptied as soon as an automatic or a manual cut is successfully completed. The new meshes originating from the new cut are then saved in the exports folder. The results list at the frontend reads from the exports folder.

A. Meshes communication (.stl)

All returned meshes coming either from a manual cut, an automatic cut or even tester meshes are automatically saved at the exports folder.

In case of normal meshes resulting from a cut, they are saved as Meshn.stl. In case of test meshes, these are saved as TestMeshn.stl. "n" represents the number of cuts in either case.

B. Vector communication (.svg)

Vector images generated from **grab2DSlice** to be used in manual cutting are saved in the exports folder as well. These are saved as slicen.svg, where "n" ranges from zero to four.

2.2.6. OS limitations

FitnSit was developed and tested mainly on Windows 10 64-bit. It is not guaranteed to work properly on other operating systems. Functional limitations are expected on posix systems due to differences in processing of C-based rtree library. A workaround on macOS is to manually install rtree using "brew install spatialindex" or using "sudo apt-get install libspatialindex-dev" on Unix-based systems.