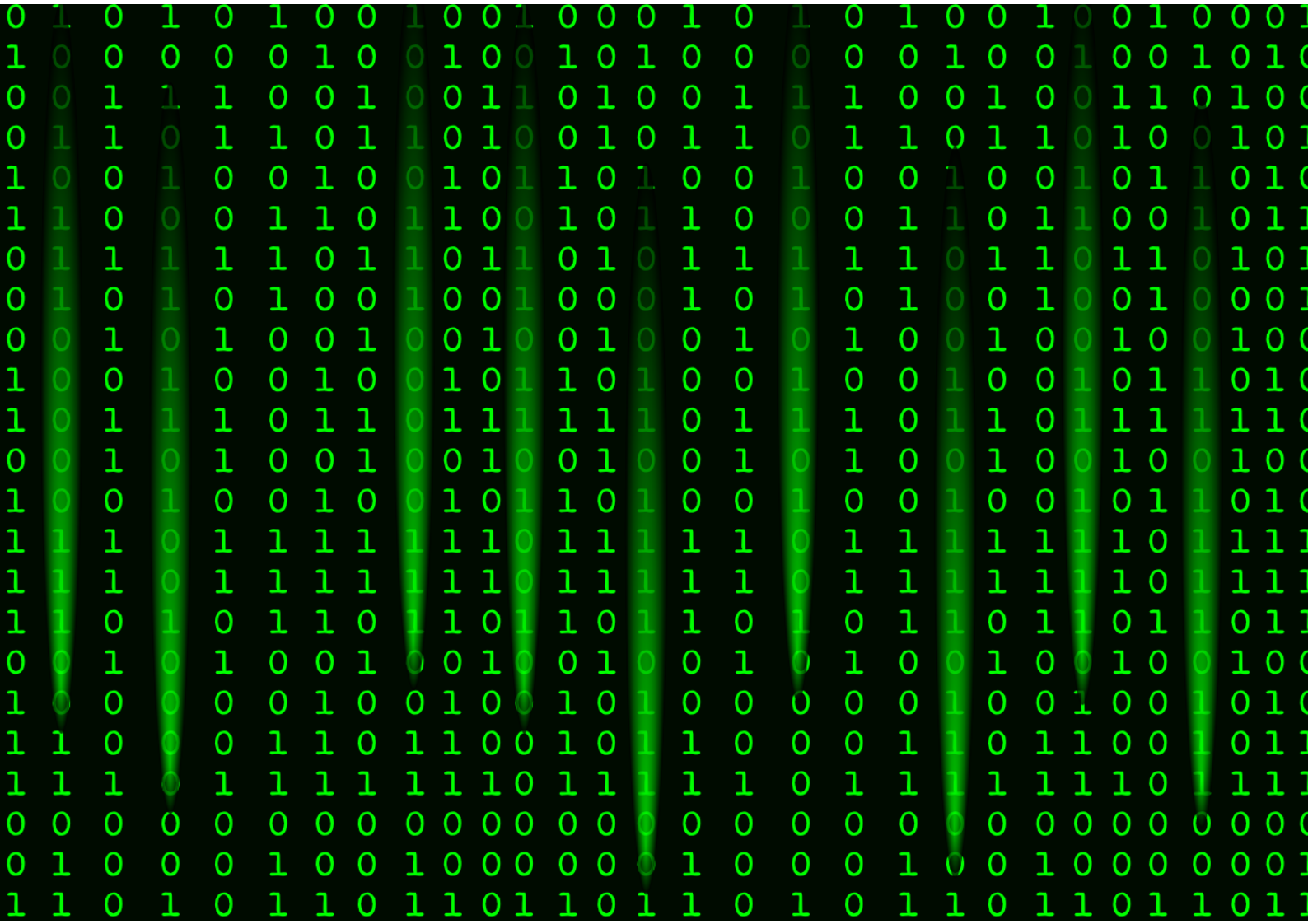
CODE DOKUMENTATION

SARLEX SERIALIZER



Autoren:

Sarah Soomro

Alexander Wensierski

Inhaltsverzeichnis

Einleitung 1
init.py 1
Plugin.json 2
PrinterInfo.py 2
SerializerUI.qml 2
Serializer.py 3

Einleitung

Insgesamt haben wir 5 Dateien, die für unser PlugIn notwendig sind:

- _init_.py
- Plugin.json
- SerializerUI.qml
- Serializer.py
- PrinterInfo.py

Darüber hinaus beinhaltet unser Repository, diverse Ressourcen, wie beispielsweise PNG-Bilder, die hier aber nicht weiter beschrieben werden.

Im Folgenden werden die 5 Dateien und die wichtigsten Bestandteile jeweils kurz genauer erläutert. Damit möchten wir einen groben Überblick über- und ein grundlegendes Verständnis für unser PlugIn bereitstellen. Auf eine detaillierte Beschreibung der Elemente der jeweiligen Module, wird an dieser Stelle - von einigen Ausnahmen abgesehen - verzichtet, da dies durch konsequentes Kommentieren, bereits im Code selbst geleistet wurde.

init.py

In dieser Datei wird das PlugIn in Cura registriert. Außerdem werden hier Metadaten unseres PlugIns zur Verfügung gestellt.

Zu Beginn [Zeile 1 - 5] erfolgen 3 Import-Statements, die für die Registrierung benötigt werden.

Im Mittelteil [Zeile 7 - 18] wird die Funktion „getMetaData()“ definiert, welche die Metadaten unseres PlugIns zurückgibt.

Zuletzt [Zeile 20 - 22] wird die Funktion „register()“ definiert, welche unser PlugIn letzten Endes bei Cura anmeldet. Hier wird dem System bereits mitgeteilt, dass es sich bei dem PlugIn um eine Extension handelt. Es fällt in diese Rubrik, denn es stellt Operationen bereit, die nur durch die Aktion eines Users ausgelöst werden können. Bei der Registrierung wird

dem System ein Serializer-Objekt zurückgegeben. Mit dem Konstruktoraufruf, werden u.a. der Menüpunkt „Serializer“ im Extensions-Reiter von Cura, sowie der Unterpunkt „Open Tool“ gesetzt.

Plugin.json

In dieser Datei werden spezielle Metadaten gespeichert, die für die Registrierung unseres PlugIns bei Cura notwendig sind.

PrinterInfo.py

In dieser Datei werden druckerspezifische Informationen zwischengespeichert und ausgelesen. Diese Informationen sind essenziell für den G-Code-Manipulationsabschnitt in der Serializer.py Datei. Da die Elemente der PrinterInfo.py Datei sehr engmaschig mit den Elementen des G-Code-Abschnitts verflochten sind, werden diese, an entsprechender Stelle im G-Code-Manipulationsabschnitt der Serializer.py Datei beschrieben.

SerializerUI.qml

In dieser Datei wird die grafische Benutzeroberfläche unseres Programms aufgebaut. Da wir hier das Framework QT benutzen, ist die Datei mit der QT Markup Language, kurz QML, geschrieben.

Zu Beginn erfolgen insgesamt 5 Import-Statements. Die ersten 3 Import-Statements werden für die Nutzung der qt-eigenen Elemente (wie z.B. Labels, Buttons und Sliders) benötigt. Die letzten beiden Import-Statements werden für die Interaktion mit dem Hauptprogramm Cura benötigt.

Die unterste Ebene der QML-Datei bildet der UM.Dialog. Er zieht sich von Zeile 11 bis Zeile 540. Alle weiteren Elemente befinden sich innerhalb des UM-Dialogs. Die besagten Elemente lassen sich in folgende 8 Blöcke unterteilen:

1. Den Property-Variablen-Block [Zeile 27 - 35]: Hier werden Variablen definiert, die für die Kommunikation mit dem Python-Backend (der Serializer.py-Datei) zwingend erforderlich sind.

2. Den Background-Image-Block [Zeile 40 - 56]. Hier wird ein Hintergrundbild definiert, welches die Bedienelemente unserer GUI untermalt und optisch aufwertet.

3. Den Label-Block [Zeile 61 - 170]. Hier werden insgesamt 7 Label definiert, die dafür benutzt werden, um Elemente unserer GUI zu beschriften bzw. um kritische Informationen auf unserer GUI ausgeben zu lassen.

4. Den Icon-Image-Block [Zeile 176 - 234]. Hier werden die Hintergrundbilder der Infill-Methoden-Buttons definiert.

5. Den Button-Block [Zeile 239 - 398] Hier werden die Buttons unserer GUI definiert. Dieser Block gliedert sich in zwei Unterblöcke:

    - Den Methoden-Button-Block [Zeile 239 - 351]: Hier werden die Buttons des Methoden-Rads definiert.

    - Und den Abschlussleisten-Button-Block [Zeile 356 - 398]: Hier werden die Buttons der Abschluss-Leiste definiert.

6. Den Slider-Block [Zeile 402 - 454]: Hier werden die beiden Slider unserer GUI definiert.

7. Den Textfield-Block [Zeile 457 - 502]: Hier werden die Textfelder unserer GUI definiert.

8. Und last but not least den Schnellzugriffs-Button-Block [Zeile 509 - 538]: Hier wird der Schnellzugriffs-Button definiert, der neben dem Slice-Button erscheint, sobald über das PlugIn eine Stückzahl eingetragen wurde.

Serializer.py

In dieser Datei wird unsere Programm-Logik festgelegt.

Zu Beginn [Zeile 1 - 19] erfolgen die Importstatements. Sie lassen sich wie folgt unterteilen:

- Imports zur Verwendung python-eigener Klassen und Funktionen [Zeile 1 - 2].

- Imports zur Verwendung von Klassen und Funktionen aus den Uranium- und Cura-Repositories [Zeile 4 - 15].

- Imports zur Kommunikation zwischen Python und QT [Zeile 17 - 19].

Es folgt die Definition unserer Serializer-Klasse, diese beginnt in Zeile 23 und zieht sich bis zum Ende der Datei. Die Serializer-Klasse lässt sich in folgende 4 Blöcke unterteilen:

1. Den Konstruktor-Block [Zeile 28 - 54]. Hier definieren wir, wie eine Instanz unserer Serializer-Klasse initialisiert wird.

2. Den System-Anbindungs-Block [Zeile 57 - 74]. Hier definieren wir, die Anbindung an das Hauptprogramm Cura, sowie die grundlegende Interaktion mit selbigen.

3. Den Slot-Block [Zeile 77 - 176]. Hier definieren wir (Hilfs-)Methoden, die wir zur Interaktion mit der Benutzeroberfläche (unserer Serializer.qml) benötigen.

4. Den G-Code-Block [Zeile 178 - 254]. Dieser Block bildet das Herzstück unseres PlugIns. Hier implementieren wir die Funktionalitäten, die dafür sorgen, dass ein

fertig-gedrucktes Objekt vom Druckkopf von der Fertigungsplatte gestoßen und die nächste Instanz des Seriendrucks in Auftrag gegeben wird. Da es sich hier um einen kriegsentscheidenden, nicht-trivialen Abschnitt handelt, wird diesem Block im Folgenden eine detaillierte Beschreibung gewidmet:

Zur Laufzeit von Cura gibt es genau eine Instanz von Application.py, die für die Hauptschleife des Programms sowie für die Erstellung weiterer zentraler Objekte zuständig ist. Über sie wird im Code auf den OutputDeviceManager zugegriffen. Sobald emittiert wird, dass in ein OutputDevice geschrieben werden soll, wird die Methode seriesprint() in dem Serializer-Objekt aufgerufen.

```
Application.getInstance().getOutputDeviceManager().writeStarted.connect(self.seriesprint)
```

Als erstes wird in seriesprint() überprüft, ob der fertig-Button gedrückt wurde. Wenn die Instanzvariable fertigclicked vom Serializer-Objekt 0 ist, ist er nicht gedrückt worden und der Seriendruck wird abgebrochen.

```
if self._fertigclicked == 0:
    return
```

Sobald der Benutzer einmal auf den fertig-Button geklickt hat, wird dies im Plugin gespeichert, indem fertigclicked auf 1 gesetzt wird. Innerhalb der Serializer.UI-Instanz wird dann die Methode update_fertig_button() aufgerufen.

```
@pyqtSlot()
def update_fertig_button(self):
    self._fertigclicked = 1
```

Steht fertigclicked auf 1, wird als nächstes der gcode zum Objekt in der lokalen Variable gcode_list gespeichert. Dazu wird zunächst das Dictionary gcode_dict von der aktuellen Scene in gcode_dict gespeichert und anschließend der Eintrag mit der ID der aktuellen Buildplate, also der gcode, in gcode_list gespeichert.

```
scene = Application.getInstance().getController().getScene()
if not hasattr(scene, "gcode_dict"):
    return
gcode_dict = getattr(scene, "gcode_dict")
if not gcode_dict:
    return
active_build_plate_id = CuraApplication.getInstance().getMultiBuildPlateModel().activeBuildPlate
gcode_list = gcode_dict[active_build_plate_id]
if not gcode_list:
    return
```

Für den Fall, dass die Stückzahl auf 0 gesetzt wurde, wird der gcode mit

```
";Stückzahl = 0! Es ist nichts zu drucken. \n"
```

überschrieben.

Andernfalls handelt es sich um einen Seriendruck. In dem ersten Abschnitt des gcodes wird als Kommentar geschrieben, wieviele Objekte gedruckt werden.

```
if self._stueckzahl > 1:
    gcode_list[0] += ";Series print: " + str(self._stueckzahl) + " objects\n"
else:
    gcode_list[0] += ";Series print: " + str(self._stueckzahl) + " object\n"
```

Weiter werden der erste und letzte Paragraph des gcodes zwischengespeichert und vorläufig aus dem gcode entfernt.

```
firstparagraph = gcode_list.pop(0)
lastparagraph = gcode_list.pop()
```

Für das folgende Herunterschrieben der Objekte von der Druckplatte werden Drucker-spezifische Angaben benötigt. Diese werden über ein PrinterInfo-Objekt ermittelt. Das Serializer-Objekt hat dazu ein Instanzvariable printerInfo.
Ein PrinterInfo-Objekt hat eine Instanzvariable printername für den Namen des aktuellen Druckers und firstparagraph für den ersten Paragraphen des gcodes.

```
class PrinterInfo():
    def __init__(self) -> None:
        self._printername = ""
        self._firstparagraph = None
```

Den Namen des aktuellen Druckers bekommt das PrinterInfo-Objekt über die updatemachine()-Funktion, die innerhalb der qml-Datei bei Drücken des fertig-Buttons oder Schließen der Serializer-Gui aufgerufen wird. Über das Serializer-Objekt wird der Name an das Printer-Info-Objekt weitergegeben.

```
@pyqtSlot(str)
def updatemachine(self, machinename):
    self._printerInfo.updatePrinterName(machinename)
```

(aus Serializer.py)

```
def updatePrinterName(self, machinename):
    self._printername = machinename
```

(aus PrinterInfo.py)

Innerhalb der qml-Datei ist machinename eine globale Variable, die den Namen des Druckers aus Cura ausliest.

```
property string machinename : Cura.MachineManager.activeMachine.id
```

Weiter in der Methode seriesprint() wird der aktuelle erste Paragraph an das PrinterInfo-Objekt übergeben.

```
self._printerInfo.updateFirstParagraph(firstparagraph)
```

Anschließend werden über dieses die gcode-Befehle pushx, pushy, pushz für das Herunterschieben des Objekts ermittelt mit den Aufrufen findpushx(),findpushy(),findpushz().

```
pushx = self._printerInfo.findpushx()
pushy = self._printerInfo.findpushy()
pushz = self._printerInfo.findpushz()
```

Bezeichne maxz die z-Koordinate des Extruders nach dem Druck sowie minx die minimale x- und maxx die maximale x-Koordinate des Extruders während des Drucks. Der Extruder soll nach dem Druck zuerst zum Punkt 0 auf der y-Achse, dann zum Punkt 0 auf der x-Achse, dann zum Punkt maxz/2, dann zum Punkt minx+(maxx-minx)/2 fahren, sodass er mittig hinter dem Objekt steht. Anschließend soll er mit minimaler Geschwindigkeit in maximale y-Ausrichtung fahren, um das Objekt herunterzuschieben.

Mit Hilfe der Kommentare in dem gcode im ersten Paragraphen ist es nun möglich maxz, minx,maxx des 3D-Drucks auszulesen. Da der gcode bei den Ultimakern des Typs 2 und 3 verschieden aufgebaut ist, wird eine Fallunterscheidung gemacht.

```
def findpushx(self):
    minx = "0"
    maxx = "0"
    textarray = self._firstparagraph.split(";")
    for element in textarray:
        if "Ultimaker 2" in self._printername:
            if "MINX" in element:
                minx = element.split(":")[1]
                ...
        elif "Ultimaker 3" in self._printername:
            if "MIN.X" in element:
                minx = element.split("MIN.X:")[-1]
```

```
            ...
    return float(minx) + (float(maxx) - float(minx))/2
```

Die Methode findpushz() ist analog aufgebaut und die Methode finspushy() gibt die maximale y-Koordinate entsprechend der Druckabmessungen der Drucker in ihren Handbüchern zurück.

```
def findpushy(self):
    if "Ultimaker 2 Go" in self._printername:
        return "120"
    elif "Ultimaker 2" in self._printername:
        return "223"
    elif "Ultimaker 3" in self._printername:
        return "215"
    else:
        return "0"
```

Weiter in seriesprint() wird das Herunterschieben von der Druckplatte schrittweise als gcode-Befehle eingefügt. Dabei werden zusätzlich die Methoden waitncool() und slowestspeed() aufgerufen.

```
gcode_list[-1] += "G0 Y0.0" + "\n"
gcode_list[-1] += "G0 X" + str(pushx) + " Y0.0" + "\n"
gcode_list[-1] += "G0 X" + str(pushx) + " Y0.0 Z" + str(pushz) + "\n"
gcode_list[-1] += self._printerInfo.waitncool()
gcode_list[-1] += "G0 X" + str(pushx) + " Y" + pushy + " Z" + str(pushz) +
self._printerInfo.slowestspeed()
```

Die Methode waitncool() sorgt dafür, dass nach dem Druck gewartet wird bis das Objekt fest ist und lässt den Drucker 5 Minuten pausieren.

```
def waitncool(self):
# G4 S300 – Warte 300 Sekunden, also 5 Minuten
    if "Ultimaker 2 Go" in self._printername:
        return "G4 S300\n"
    elif "Ultimaker 2" in self._printername:
        return "G4 S300\n"
    elif "Ultimaker 3" in self._printername:
# M190 - Druckbett abkühlen auf 0°C und dabei warten
        return "M190 R0\nG4 S300\n"
    else:
        return ";"
```

Die Methode slowestspeed gibt die minimale Geschwindigkeit des Druckers als gcode-Befehl wieder.

```
def slowestspeed(self):
    # 30 mm/s, sind 1800 mm/m
    return " F1800\n"
```

Als nächstes wird der gcode entsprechend der Stückzahl vervielfacht und als Kommentar der Index eines Objekts vor jedem Druck geschrieben.

```
copygcode = gcode_list.copy()
gcode_list[0] += "; 1.object\n"
for i in range(1, self._stueckzahl):
    gcode_list = gcode_list + [";" + str(i+1) + ". object\n"]
    gcode_list = gcode_list + copygcode
```

Als letztes werden der erste und letzte Paragraph wieder zum gcode hinzugefügt.

```
gcode_list = [firstparagraph] + gcode_list
gcode_list = gcode_list + [lastparagraph]
```

Und danach der manipulierte G-Code an das System übergeben.

```
gcode_dict[active_build_plate_id] = gcode_list
setattr(scene, "gcode_dict", gcode_dict)
```

Da es nur möglich ist, den gcode direkt für d Ultimaker 2 und 3 auszulesen, ist das Plugin auf diese eingeschränkt. PrinterInfo ist so aufgebaut, dass die Klasse um weitere Drucker ergänzt werden kann.