



Source Code

"""

OnCue Projector

Copyright 2017 Andrew Wong <featherbear@navhaxs.au.eu.org>

The following code is licensed under the GNU Public License Version v3.0  
"""

# File: OnCue.py

\_\_VERSION\_\_ = "0.0.0.1"

if \_\_name\_\_ == "\_\_main\_\_":

*# Import modules*

import glob

import os

import platform

import re

import sys

import tempfile

import win32api

import winreg

import subprocess

import win32com.client

from PyQt5 import QtCore, QtGui, QtWidgets

*# Import the rest of the OnCue files*

import oncue

*# Expose modules and functions to global scope*

Vlc = oncue.lib.vlc

vlc = Vlc.Instance()

colorPicker = lambda o: oncue.forms.colorPicker.colorPicker(o, theme=states["interface"]["theme"]).exec\_()

dprint = oncue.lib.utils.dprint

selector = None

class Application(QtWidgets.QMainWindow):

"""

Container class

"""

def \_\_init\_\_(self):

"""

Set up the interface

"""

super(Application, self).\_\_init\_\_()

global selector

self.setWindowTitle("OnCue")

self.setWindowIcon(QtGui.QIcon("OnCue.ico"))

self.resize(900, 900)

self.setMinimumSize(self.size())

self.setMaximumSize(self.size())

self.setWindowFlags(QtCore.Qt.FramelessWindowHint)

selector = QtWidgets.QStackedWidget()

self.setCentralWidget(selector)

states["interface"]["ui"] = [MAIN(), PREFERENCES()]

[selector.addWidget(ui) for ui in states["interface"]["ui"]]

class MAIN(QtWidgets.QWidget, oncue.forms\_gen.main.Ui\_OnCue):

"""

Main Interface

"""

def \_\_init\_\_(self):

"""

Initialise class

"""

QtWidgets.QWidget.\_\_init\_\_(self)

self.setupUi(self)

output.PPTevents.updateSlide = self.powerpointSlides.setCurrentRow

self.contentControls.setCurrentIndex(0)

*# Update video scrubber position*

class ProgressBarUpdater(QtCore.QThread):

tick = QtCore.pyqtSignal(int)

def run(self):

while True:

self.sleep(1)

self.tick.emit(oncue.lib.utils.confine(int(output.VLCposition() \* 1000), 0, 1000))

self.mediaProgressBarThread = ProgressBarUpdater()

self.mediaProgressBarThread.tick.connect(

lambda value: self.mediaProgressBar.setValue(value) or self.mediaProgressBar.repaint())

def validateClick(self, pos, obj, callback):

validation = bool(obj.itemAt(pos if isinstance(pos, QtCore.QPoint) else pos.pos()))

callback() if validation else False

def playItem(self):

"""

Plays the selected item

"""

data = self.listItemsPrimary.currentItem().data(256)

output.load(data)

if data["type"] == "media":

*# Plays video*

self.mediaProgressBarThread.start()

self.mediaProgressBarThread.setPriority(QtCore.QThread.TimeCriticalPriority)

self.mediaControls\_PLAY.click()

self.contentControls.setCurrentIndex(2)

elif data["type"] == "powerpoint":

*# Clear existing content in the slide preview list*

self.powerpointSlides.clear()

*# Connect to PowerPoint COM*

PPTApplication = win32com.client.Dispatch("PowerPoint.Application")

Presentation = PPTApplication.Presentations.Open(data["path"].replace("/", "\\"),

WithWindow=False)

*# Create slide previews*

temp = tempfile.TemporaryDirectory().name

Presentation.Export(temp, "png")

i = 1

for file in glob.iglob(temp + "\\\*.PNG"):

item = QtWidgets.QListWidgetItem()

item.setIcon(QtGui.QIcon(file))

item.setText(str(i))

item.setTextAlignment(QtCore.Qt.AlignCenter)

i += 1

self.powerpointSlides.addItem(item)

self.contentControls.setCurrentIndex(1)

else:

*# 'unknown' case - Hide controls*

self.contentControls.setCurrentIndex(0)

def createQListWidgetItem(self, data):

"""

Creates a QListWidgetItem() instance given a file path

"""

item = QtWidgets.QListWidgetItem()

path = data.toLocalFile()

type = oncue.lib.utils.identifyFileType(path)

item.setText(data.fileName())

item.setData(256, {

'type': type,

'path': path

})

if type == "media":

item.setToolTip(oncue.lib.utils.parseMedia(path))

else:

item.setToolTip("Path: " + path)

return item

def handleDropEvent(self, e):

"""

Handle adding items into the playlist

"""

if e.mimeData().hasFormat('application/x-qabstractitemmodeldatalist'):

return QtWidgets.QListWidget.dropEvent(self.listItemsPrimary, e)

data = e.mimeData().urls()[0]

if isinstance(data, QtCore.QUrl): self.listItemsPrimary.addItem(self.createQListWidgetItem(data=data))

def setupUi(self, MAIN):

"""

Register button functions

"""

super().setupUi(MAIN)

self.btnClear.clicked.connect(lambda: output.clear() or self.contentControls.setCurrentIndex(0))

self.btnSettings.clicked.connect(

lambda: states["interface"]["ui"][1].updateSettingInterface() or selector.setCurrentIndex(1))

self.btnExit.clicked.connect(lambda: dprint("Quitting") or output.PPTclose() or sys.exit(0))

*# Output controls*

[self.btnOutput.itemAt(button).widget().setStyleSheet("") for button in range(self.btnOutput.count())]

if len(states["screens"]) == 1:

dprint("Output disabled, disabling display buttons")

self.btnOutput\_wrap.setEnabled(False)

else:

self.btnOutputClear.clicked.connect(output.contentHide)

self.btnOutputContent.clicked.connect(output.contentShow)

self.btnOutputDesktop.clicked.connect(output.contentDesktop)

*# Playlist right-click menu*

self.listItemsPrimary.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)

self.listItemsPrimary.dragEnterEvent = lambda e: (

e.accept() if e.mimeData().hasUrls() or e.mimeData().hasFormat(

'application/x-qabstractitemmodeldatalist') else e.ignore())

self.listItemsPrimary.dropEvent = self.handleDropEvent

self.listItemsPrimary.customContextMenuRequested.connect(

lambda \_: self.validateClick(\_, self.listItemsPrimary, lambda: self.contextMenu.exec\_()))

*# Playlist doubleclick*

self.listItemsPrimary.mouseDoubleClickEvent = lambda \_: self.validateClick(\_, self.listItemsPrimary,

self.playItem)

*# Powerpoint doubleclick*

self.powerpointSlides.mouseDoubleClickEvent = lambda \_: self.validateClick(\_, self.powerpointSlides,

lambda: output.PPTslide(

self.powerpointSlides.currentRow() + 1))

*# Powerpoint controls*

self.powerpointControls\_PREVIOUS.clicked.connect(output.PPTprevious)

self.powerpointControls\_NEXT.clicked.connect(output.PPTnext)

*# Media controls*

self.mediaControls\_PAUSE.clicked.connect(output.VLCpause)

self.mediaControls\_PLAY.clicked.connect(output.VLCplay)

self.mediaControls\_MUTE.clicked.connect(lambda: output.VLCmute(self.mediaControls\_MUTE.isChecked()))

*# Update interface theme*

inactive, checked, hover = states["interface"]["theme"]

self.theming.setStyleSheet(

"QLabel[objectName='Header'] {background: #%s} "

"QPushButton {background-color: #%s} "

"QPushButton:checked {background-color: #%s} "

"QPushButton:hover:!checked {background-color: #%s}"

"QProgressBar:chunk {background-color: #%s;}"

"QLabel[objectName='mediaProgressSeek'] {border-left-color: #%s;}"

% (checked, inactive, checked, hover, checked, hover))

class CCMenu(QtWidgets.QMenu):

"""

Right-click menu

"""

def \_\_init\_\_(self, area, parent):

"""

Menu entries

"""

super().\_\_init\_\_()

self.area = area

self.parent = parent

self.add("Play", self.play)

self.addSeparator()

*# self.add("Debug", self.debug)*

self.add("Remove", self.remove)

def add(self, label: str, function):

*# helper function to create menu entries*

item = QtWidgets.QAction(label, self)

item.triggered.connect(function)

self.addAction(item)

def exec\_(self):

self.item = self.area.currentItem()

*# Open right-click menu where the cursor is*

super().exec(QtGui.QCursor.pos())

def play(self):

*# Play item*

self.parent.playItem()

def remove(self):

*# Remove item from playlist*

self.area.takeItem(self.area.currentRow())

*# Register right-click menu*

self.contextMenu = CCMenu(self.listItemsPrimary, self)

class PREFERENCES(QtWidgets.QWidget, oncue.forms\_gen.settings.Ui\_Settings):

"""

Preferences interface

"""

def \_\_init\_\_(self):

QtWidgets.QWidget.\_\_init\_\_(self)

self.setupUi(self)

self.viewTabs\_General.click() *# Set General tab (Index 0) as default*

*# Populate monitor dropdowns*

displaysArray = ["%s. %s" % (str(i), states["screens"][i]['name']) for i in states["screens"]]

self.prefOutputDisplayID.addItems(displaysArray)

*# Set component version info*

self.aboutVersions.setText(

"\n".join(["%s Version: %s" % (

component, states["versions"][component] if states["versions"][component] else "n/a") for component in

states["versions"].keys()]))

*# Set custom and last states*

self.customtheme = states["interface"]["theme"]

self.lastbutton = themes.index(states["interface"]["theme"]) if states["interface"][

"theme"] in themes else self.prefTheme.count() - 1

self.customoutputbackground = states["display"]["outputbackground"]

self.lastoutputbackground = backgrounds.index(states["display"]["outputbackground"]) if states["display"][

"outputbackground"] in backgrounds else self.prefOutputBackground.count() - 1

self.updateSettingInterface()

def setupUi(self, SettingsWindow):

"""

Connect buttons to functions

"""

super().setupUi(SettingsWindow)

[self.viewTabsHeader.itemAt(button).widget().clicked.connect(self.handleTabSwitch) for button in

range(self.viewTabsHeader.count())]

self.btnBack.clicked.connect(self.back)

self.btnSave.clicked.connect(self.handleSave)

*# GENERAL*

[self.prefTheme.itemAt(buttonID).widget().clicked.connect(self.handleTheme\_standard) for buttonID in

range(self.prefTheme.count() - 1)]

self.prefTheme\_CUSTOM.clicked.connect(self.handleTheme\_custom)

[self.prefOutputBackground.itemAt(buttonID).widget().clicked.connect(self.handleOutputBackground\_standard) for

buttonID

in range(self.prefOutputBackground.count() - 1)]

self.prefOutputBackground\_CUSTOM.clicked.connect(self.handleOutputBackground\_custom)

self.prefOutputDisplayID.currentIndexChanged.connect(self.setUnsaved)

self.prefBackgroundMedia\_OFF.clicked.connect(lambda: self.handleEnableBackgroundMedia(False))

self.prefBackgroundMedia\_ON.clicked.connect(lambda: self.handleEnableBackgroundMedia(True))

[self.prefBackgroundAudio.itemAt(button).widget().clicked.connect(self.setUnsaved) for button in

range(self.prefBackgroundAudio.count())]

self.btnSystemSettings.clicked.connect(lambda: subprocess.Popen('desk.cpl', shell=True))

"""

Change theme schemes

"""

def handleTheme\_standard(self):

self.lastbutton = [self.prefTheme\_BLUE.isChecked(), self.prefTheme\_RED.isChecked(),

self.prefTheme\_GREY.isChecked(),

self.prefTheme\_DARK.isChecked()].index(True)

self.setUnsaved()

def handleTheme\_custom(self):

"""

"""

inactive, checked, hover = self.customtheme

result = colorPicker({"Inactive": inactive, "Checked": checked, "Hover": hover})

if result:

self.customtheme = tuple(result.values())

self.lastbutton = 4

self.setUnsaved()

else:

self.prefTheme.itemAt(self.lastbutton).widget().setChecked(True)

def handleOutputBackground\_standard(self):

self.lastoutputbackground = [self.prefOutputBackground\_BLACK.isChecked(),

self.prefOutputBackground\_WHITE.isChecked()].index(True)

self.setUnsaved()

def handleOutputBackground\_custom(self):

result = colorPicker(self.customoutputbackground)

if result:

self.customoutputbackground = result

self.lastoutputbackground = 2

self.setUnsaved()

else:

self.prefOutputBackground.itemAt(self.lastoutputbackground).widget().setChecked(True)

def back(self):

"""

Handle back button presses

"""

*# Check for saved changes and prompt about lost changes*

if self.btnSave.isEnabled():

result = MODAL("Changes were made.\n\nSave?").exec\_()

if result == -1:

return

elif not result:

self.updateSettingInterface()

elif not self.handleSave():

return

selector.setCurrentIndex(0)

def updateSettingInterface(self):

"""

Update control states of preference control elements

"""

*# GENERAL*

[self.prefTheme.itemAt(buttonID).widget().setChecked for buttonID in range(self.prefTheme.count())][

themes.index(states["interface"]["theme"]) if states["interface"][

"theme"] in themes else self.prefTheme.count() - 1](

True)

[self.prefOutputBackground.itemAt(buttonID).widget().setChecked for buttonID in

range(self.prefOutputBackground.count())][

backgrounds.index(states["display"]["outputbackground"]) if states["display"][

"outputbackground"] in backgrounds else self.prefOutputBackground.count() - 1](

True)

*# DISPLAY*

self.prefOutputDisplayID.setCurrentIndex(states["display"]["outputID"] - 1)

self.handleEnableBackgroundMedia(bool(states["display"]["backgroundmedia"]))

self.prefBackgroundMedia.itemAt(states["display"]["backgroundmedia"]).widget().setChecked(True)

self.prefBackgroundAudio.itemAt(states["display"]["backgroundaudio"]).widget().setChecked(True)

inactive, checked, hover = states["interface"]["theme"]

self.theming.setStyleSheet(

"QLabel[objectName='Header'] {background: #%s} "

"QPushButton {background-color: #%s} "

"QPushButton:checked {background-color: #%s} "

"QPushButton:hover:!checked {background-color: #%s}" % (checked, inactive, checked, hover))

self.btnSave.setEnabled(False)

def setUnsaved(self):

"""

Set unsaved state (by enabling the save button)

"""

self.btnSave.setEnabled(True)

def handleTabSwitch(self):

"""

Handle tab switching events

"""

self.viewTabs.setCurrentIndex(

[self.viewTabs\_General.isChecked(), self.viewTabs\_Display.isChecked(), self.viewTabs\_Remote.isChecked(),

self.viewTabs\_About.isChecked()].index(True))

def handleEnableBackgroundMedia(self, state):

"""

Enables/Disables the background audio behaviour feature

"""

state = not state

self.setUnsaved()

self.prefBackgroundAudio\_LABEL.setEnabled(state)

[self.prefBackgroundAudio.itemAt(buttonID).widget().setEnabled(

state) for buttonID in

range(self.prefBackgroundAudio.count())]

@staticmethod

def saveChanges(key, value):

"""

Save settings to registry and global store

"""

parent, name = key.split("/")

states[parent][name] = value

QSettings.setValue(key, list(value) if isinstance(value, tuple) else value)

def handleSave(self):

"""

Saves configuration settings

"""

*# Interface theme*

themeID = [self.prefTheme.itemAt(button).widget().isChecked() for button in

range(self.prefTheme.count())].index(True)

self.saveChanges("interface/theme", self.customtheme if themeID == 4 else themes[themeID])

*# Output background colour*

outputbackgroundID = [self.prefOutputBackground.itemAt(button).widget().isChecked() for button in

range(self.prefOutputBackground.count())].index(True)

self.saveChanges("display/outputbackground",

self.customoutputbackground if outputbackgroundID == 2 else backgrounds[outputbackgroundID])

*# Save background media behaviour*

self.saveChanges("display/backgroundmedia", 1 if self.prefBackgroundMedia\_ON.isChecked() else 0)

self.saveChanges("display/backgroundaudio", 1 if self.prefBackgroundAudio\_ON.isChecked() else 0)

*# Output monitor*

if states["display"]["outputID"] != self.prefOutputDisplayID.currentIndex() + 1:

self.saveChanges("display/outputID", self.prefOutputDisplayID.currentIndex() + 1)

self.saveChanges("remote/apienabled", 1 if self.prefRemoteAPI\_ON.isChecked() else 0)

*# Update interface theme*

inactive, checked, hover = states["interface"]["theme"]

for ui in states["interface"]["ui"]:

ui.theming.setStyleSheet(

"QLabel[objectName='Header'] {background: #%s} "

"QPushButton {background-color: #%s} "

"QPushButton:checked {background-color: #%s} "

"QPushButton:hover:!checked {background-color: #%s}" % (checked, inactive, checked, hover))

*# Reflect changed settings*

self.updateSettingInterface()

*# Update output windows*

output.draw()

*# stage.draw() # TODO not implemented*

dprint("Saved settings")

self.btnSave.setEnabled(False)

return True

class MODAL(QtWidgets.QDialog, oncue.forms\_gen.modal.Ui\_modal\_ynoc):

"""

Message box

"""

def \_\_init\_\_(self, text, \*\*kwargs):

QtWidgets.QDialog.\_\_init\_\_(self)

self.setupUi(self, text, \*\*kwargs)

def setupUi(self, MODAL, text, yes=True, no=True, cancel=True, ok=False, tool=True):

super().setupUi(MODAL)

self.setWindowFlags(QtCore.Qt.FramelessWindowHint)

if tool: self.setWindowFlags(self.windowFlags() | QtCore.Qt.Tool)

*# Change response button visibility*

if not ok: self.response.button(QtWidgets.QDialogButtonBox.Ok).hide()

self.response.button(QtWidgets.QDialogButtonBox.Yes).clicked.connect(

lambda: self.setResult(1)) if yes else self.response.button(QtWidgets.QDialogButtonBox.Yes).hide()

self.response.button(QtWidgets.QDialogButtonBox.No).clicked.connect(

lambda: self.setResult(0)) if no else self.response.button(QtWidgets.QDialogButtonBox.No).hide()

self.response.button(QtWidgets.QDialogButtonBox.Cancel).clicked.connect(

lambda: self.setResult(-1)) if cancel else self.response.button(QtWidgets.QDialogButtonBox.Cancel).hide()

self.message.setText(text)

*# Set theming*

[btn.setStyleSheet(

"QPushButton {background-color: #BAB9BA; \_background-color: #%s;} "

"QPushButton:pressed {background-color: #%s} "

"QPushButton:hover:!pressed {background-color: #%s}" % states["interface"]["theme"]) for btn in

self.response.buttons()]

*# Create Qt Application instance*

app = QtWidgets.QApplication(sys.argv)

app.setAttribute(QtCore.Qt.AA\_EnableHighDpiScaling)

*# Constants*

themes = [("bab9ba", "509df3", "8cc5ff"), ("d8d8d8", "ff6666", "ffadad"), ("d8d8d8", "808080", "bcbcbc"),

("d8d8d8", "000000", "494949")]

backgrounds = ["000000", "ffffff"]

*# Read preferences from registry*

QSettings = QtCore.QSettings("featherbear", "OnCue Projector")

states = {

'display': {},

'screens': {},

'interface': {

'ui': [],

'theme': ()

},

'remote': {

'apienabled': 0,

},

'versions': {}

}

states["display"]["outputID"] = QSettings.value("display/outputID", 2)

theme = QSettings.value("interface/theme", 0)

states["interface"]["theme"] = tuple(QSettings.value("interface/theme", themes[0]))

states["display"]["outputbackground"] = str(QSettings.value("display/outputbackground", backgrounds[0]))

states["display"]["backgroundmedia"] = QSettings.value("display/backgroundmedia", 0)

states["display"]["backgroundaudio"] = QSettings.value("display/backgroundaudio", 0)

*# Verify OS is Windows*

platform = platform.system()

print("OS is " + platform)

if platform != "Windows": MODAL("Detected OS is not Windows. Program aborting", yes=False, no=False, cancel=False, ok=True, tool=False).exec\_(); sys.exit("os")

*# Find latest version of PowerPoint present*

states["versions"]["PowerPoint"], pptregistry = None, None

try:

key = winreg.OpenKey(winreg.HKEY\_CURRENT\_USER, "Software\\Microsoft\\Office")

for version in sorted(

list(filter(lambda s: "." in s, [winreg.EnumKey(key, i) for i in range(winreg.QueryInfoKey(key)[0])])),

reverse=True, key=lambda s: float(s)):

try:

pptregistry = winreg.OpenKey(winreg.HKEY\_CURRENT\_USER,

"Software\\Microsoft\\Office\\" + version + "\\PowerPoint\\Options",

access=winreg.KEY\_ALL\_ACCESS)

states["versions"]["PowerPoint"] = version

except:

continue

except WindowsError:

pass

*# Find versions of other components*

states["versions"]["Python"] = re.search(r'\(v(.+?),', sys.version).group(1)

states["versions"]["Qt"] = QtCore.QT\_VERSION\_STR

states["versions"]["PyQt"] = QtCore.PYQT\_VERSION\_STR

states["versions"]["libVLC"] = Vlc.libvlc\_get\_version().decode('ascii')

states["versions"]["OnCue"] = \_\_VERSION\_\_

*# Enumerate display monitors*

monitors = dict([((monitor["Monitor"][0], monitor["Monitor"][1]),

(monitor["Device"], win32api.EnumDisplayDevices(monitor["Device"]).DeviceString)) for monitor in

[win32api.GetMonitorInfo(display[0]) for display in win32api.EnumDisplayMonitors()]])

*# Populate monitor information*

for i in range(app.desktop().screenCount()):

screen = app.desktop().screenGeometry(i)

topleft = (screen.left(), screen.top())

states["screens"][i + 1] = {

'width': screen.width(),

'height': screen.height(),

'physical': monitors[topleft][0],

'name': monitors[topleft][1],

}

dprint("Starting OnCue")

*# This application is to be used in a multi-monitor configuration - Output is disabled for single monitor setups*

if len(states["screens"]) == 1:

errmsg = "Only one screen detected. Outputs disabled"

dprint(errmsg)

MODAL(errmsg, yes=False, no=False, cancel=False, ok=True, tool=False).exec\_()

*# Create output screens*

dprint("Creating output screens")

output = oncue.displays.displayOutput.displayOutput(

dict(dprint=dprint, states=states, app=app, pptregistry=pptregistry, confine=oncue.lib.utils.confine, Vlc=Vlc))

output.draw()

*# Execute OnCue*

window = Application()

window.show()

sys.exit(app.exec\_())

# File: oncue/lib/utils.py

def confine(n, m, M):

"""

Confines a value inside a range

:param n: value

:param m: min

:param M: max

"""

return max(min(M, n), m)

def dprint(\*args, level=0):

"""

Print helper

"""

if DEBUG:

print(datetime.now().strftime('%H:%M:%S.%f')[:-3], "|", "DEBUG", "|", " " \* level, \*args)

def fourcc(dec):

"""

Convert a 4 byte ASCII code into a string

"""

dec = int(dec)

return chr((dec & 0XFF)) + chr((dec & 0XFF00) >> 8) + chr((dec & 0XFF0000) >> 16) + chr((dec & 0XFF000000) >> 24)

def parseMedia(path):

"""

Parse media information of a file

"""

*# Open file*

media = vlc.media\_new(path)

media.parse()

*# Get metadata*

\_title = media.get\_meta(0)

\_artist = media.get\_meta(1)

\_album = media.get\_meta(4)

*# Calculate duration*

m, s = divmod(int(media.get\_duration() / 1000), 60)

h, m = divmod(m, 60)

\_duration = ("%s:" if h else "") + "%02d:%02d" % (m, s)

*# Check for audio codec information*

\_acodec = None

tracks = list(filter(lambda track: track.type == Vlc.TrackType.audio, media.tracks\_get()))

if len(tracks) > 0:

\_acodec = fourcc(tracks[0].codec)

\_acodec2 = fourcc(tracks[0].original\_fourcc)

*# Check for video codec information*

\_vcodec = None

tracks = list(filter(lambda track: track.type == Vlc.TrackType.video, media.tracks\_get()))

if len(tracks) > 0:

\_vcodec = fourcc(tracks[0].codec)

\_vcodec2 = fourcc(tracks[0].original\_fourcc)

return ("Title: %s\n" % \_title if \_title else "") + ("Artist: %s\n" % \_artist if \_artist else "") + (

"Album: %s\n" % \_album if \_album else "") + ("Duration: %s\n" % \_duration) + (

"Audio Codec: %s%s\n" % (

\_acodec, " (%s)" % \_acodec2 if \_acodec != \_acodec2 else "") if \_acodec else "") + (

"Video Codec: %s%s\n" % (

\_vcodec, " (%s)" % \_vcodec2 if \_vcodec != \_vcodec2 else "") if \_vcodec else "") + (

"File Path: %s\n" % path)[:-1]

def identifyFileType(path):

"""

Attempts to identify the file type of a given file

"""

*# Is it a media? If it is it will have a duration*

media = vlc.media\_new(path)

media.parse()

if media.get\_duration() > 0:

return "media"

*# Check against regex patterns*

matchPatterns = {

'powerpoint': '^pp[ts]x?$',

}

extension = os.path.splitext(path)[1][1:]

for type in matchPatterns:

if re.match(matchPatterns[type], extension):

return type

return "unknown"

*# Imports*

DEBUG = True

from datetime import datetime

import sys, os, re

sys.path.insert(0, os.path.dirname(os.path.abspath(\_\_file\_\_)))

import vlc as Vlc

try:

vlc = Vlc.Instance()

except:

parseMedia = lambda \_: ""

identifyFileType = lambda \_: "unknown"

# File: oncue/forms/colorPicker.py

from PyQt5 import QtCore, QtWidgets

class customQColorDialog(QtWidgets.QColorDialog):

"""

Color picker override

"""

def \_\_init\_\_(self):

QtWidgets.QColorDialog.\_\_init\_\_(self)

self.setOption(QtWidgets.QColorDialog.NoButtons)

self.children()[10].children()[16].setText("&Hex:")

[self.children()[1].setParent(None) for elem in range(7)] *# Remove elements 1-7*

self.updateColor()

*# Elements 0, 8, 9, 10, 11 are important*

def updateColor(self, colorHex="FFFFFF"):

foc = self.children()[3].children()[17]

foc.clear()

foc.insert("#" + colorHex)

def getColor(self):

return self.children()[3].children()[17].text()[1:]

class colorPicker(QtWidgets.QDialog):

def \_\_init\_\_(self, colorDict, theme=("d8d8d8", "808080", "bcbcbc")):

QtWidgets.QDialog.\_\_init\_\_(self)

self.colorDict = colorDict

self.theme = theme

self.colordialog = customQColorDialog()

self.setupUi(self)

self.setWindowTitle("Colour Picker")

def setupUi(self, colorPicker):

"""

Interface setup

"""

class entryButton(QtWidgets.QPushButton):

def \_\_init\_\_(self, text, parent):

"""

Create category button

"""

self.parent = parent

self.identifier = text

QtWidgets.QPushButton.\_\_init\_\_(self)

self.setSizePolicy(QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum, QtWidgets.QSizePolicy.Minimum))

self.setCheckable(True)

self.setAutoExclusive(True)

self.setText(text)

self.hexValue = self.parent.colorDict[text]

self.toggled.connect(self.handleChange)

def handleChange(self, state):

if state:

self.parent.activeButton = self

self.parent.colordialog.updateColor(self.parent.colorDict[self.identifier])

else:

self.parent.colorDict[self.identifier] = self.parent.colordialog.getColor()

self.setWindowModality(QtCore.Qt.ApplicationModal)

self.setWindowFlags(QtCore.Qt.FramelessWindowHint)

self.setObjectName("colorPicker")

self.resize(900, 480)

self.setMinimumSize(self.size())

self.setMaximumSize(self.size())

self.setStyleSheet("QWidget {\n"

"text-align: center;\n"

"color: black;\n"

"border: none;\n"

"text-decoration: none;\n"

"}")

self.theming = QtWidgets.QFrame(self)

self.theming.setGeometry(QtCore.QRect(0, 0, 900, 480))

self.theming.setStyleSheet("QPushButton {background-color: #%s; color: white;}"

"QPushButton:hover {background-color: #%s;}"

"QPushButton:checked {background-color: #%s}" % self.theme)

self.horizontalLayoutWidget = QtWidgets.QWidget(self.theming)

self.horizontalLayoutWidget.setGeometry(QtCore.QRect(0, 0, 840, 480))

self.horizontalLayout = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget)

self.horizontalLayout.setContentsMargins(0, 0, 0, 0)

self.horizontalLayout.setSpacing(0)

if not isinstance(self.colorDict, str) and len(self.colorDict) != 0:

self.verticalLayout = QtWidgets.QVBoxLayout()

[self.verticalLayout.addWidget(entryButton(label, self)) for label in self.colorDict]

self.verticalLayout.itemAt(0).widget().click()

self.horizontalLayout.addLayout(self.verticalLayout)

self.horizontalLayout.setStretch(1, 1)

else:

self.colordialog.updateColor(self.colorDict)

self.horizontalLayout.addWidget(self.colordialog)

self.response = QtWidgets.QDialogButtonBox(self.theming)

self.response.setGeometry(QtCore.QRect(755, 455, 81, 20))

self.response.setOrientation(QtCore.Qt.Horizontal)

self.response.setStandardButtons(QtWidgets.QDialogButtonBox.Cancel | QtWidgets.QDialogButtonBox.Ok)

self.response.accepted.connect(self.OK)

self.response.rejected.connect(self.reject)

QtCore.QMetaObject.connectSlotsByName(self)

def OK(self):

if isinstance(self.colorDict, str):

self.colorDict = self.colordialog.getColor()

else:

self.activeButton.handleChange(False)

self.accept()

def exec\_(self):

return self.colorDict if super().exec\_() else False

# File: oncue/displays/displayOutput.py

*# Imports*

import winreg

import win32com.client

from PyQt5 import QtCore, QtWidgets

class displayOutput(QtWidgets.QWidget):

"""

Output display class

"""

def \_\_init\_\_(self, components: dict):

*# Initialise class*

self.dprint = components["dprint"]

self.states = components["states"]

self.app = components["app"]

self.pptregistry = components["pptregistry"]

self.confine = components["confine"]

self.Vlc = components["Vlc"]

self.vlc = self.Vlc.Instance()

self.dprint("Starting output display")

super(displayOutput, self).\_\_init\_\_(None, QtCore.Qt.WindowStaysOnTopHint | QtCore.Qt.Tool)

gridLayout = QtWidgets.QGridLayout(self)

gridLayout.setContentsMargins(0, 0, 0, 0)

gridLayout.setSpacing(0)

self.foreground = QtWidgets.QWidget(self)

gridLayout.addWidget(self.foreground)

self.player = self.vlc.media\_player\_new()

*# Initialise variables*

self.VLCmedia = None

self.type = None

self.VLCpaused = None

self.VLCpaused\_c = None

self.VLCmuted\_c = None

self.PPTapplication = None

self.PPTpresentation = None

self.screen = None

self.overlay = QtWidgets.QWidget(None, QtCore.Qt.WindowStaysOnTopHint)

self.overlay.setWindowOpacity(0)

self.draw()

""" Not using VLC's event manager, it seems very buggy """

*# eventmanager = self.player.event\_manager()*

*# eventmanager.event\_attach(Vlc.EventType.MediaPlayerPositionChanged, states["mediaSignals"]["update"])*

*# eventmanager.event\_attach(Vlc.EventType.MediaPlayerPaused, states["mediaSignals"]["pause"])*

*# eventmanager.event\_attach(Vlc.EventType.MediaPlayerPlaying, states["mediaSignals"]["play"])*

*# eventmanager.event\_attach(Vlc.EventType.MediaPlayerEndReached, states["mediaSignals"]["finish"])*

def draw(self):

"""

Create output window

"""

if len(self.states["screens"]) != 1:

self.hide()

self.setStyleSheet("background: #" + self.states["display"]["outputbackground"])

self.screen = self.app.desktop().screenGeometry(self.states["display"]["outputID"] - 1)

self.setGeometry(self.screen)

super(displayOutput, self).showFullScreen()

def clear(self, bypass=False):

"""

Clear output content

"""

if not bypass and self.type == "powerpoint":

self.PPTpresentation.Close()

self.overlay.hide()

self.VLCstop()

self.type = None

self.VLCpaused = None

self.VLCpaused\_c = None

def PPTclose(self):

"""

Closes the presentation

"""

try:

self.PPTpresentation.Close()

except:

pass

def PPTslide(self, slide: int):

"""

Change current slide

"""

try:

self.PPTpresentation.SlideShowWindow.View.GotoSlide(slide)

except:

pass

def PPTnext(self):

"""

Go to next slide

"""

try:

self.PPTpresentation.SlideShowWindow.View.Next()

except:

pass

def PPTprevious(self):

"""

Go to previous slides

"""

try:

self.PPTpresentation.SlideShowWindow.View.Previous()

except:

pass

class PPTevents:

"""

Handle PowerPoint COM events

"""

updateSlide = None

def OnSlideShowNextSlide(self, s):

self.updateSlide(s.View.CurrentShowPosition - 1)

def load(self, data):

"""

Prepare the content for display

"""

self.type = data["type"]

if self.type == "powerpoint":

if not self.pptregistry: return False

*# https://mail.python.org/pipermail/python-win32/2012-July/012471.html*

self.PPTapplication = win32com.client.DispatchWithEvents("PowerPoint.Application", self.PPTevents)

try:

self.PPTpresentation = self.PPTapplication.Presentations.Open(data["path"].replace("/", "\\"),

WithWindow=False)

*# Change PowerPoint output monitor setting (Touch and revert)*

reset = []

try:

reset.append((winreg.QueryValueEx(self.pptregistry, "UseAutoMonSelection")[0],

lambda value: winreg.SetValueEx(self.pptregistry, "UseAutoMonSelection", 0,

winreg.REG\_DWORD,

value)))

except WindowsError:

reset.append((None, lambda \_: winreg.DeleteValue(self.pptregistry, "UseAutoMonSelection")))

try:

reset.append((winreg.QueryValueEx(self.pptregistry, "DisplayMonitor")[0],

lambda value: winreg.SetValueEx(self.pptregistry, "DisplayMonitor", 0, winreg.REG\_SZ,

value)))

except WindowsError:

reset.append((None, lambda \_: winreg.DeleteValue(self.pptregistry, "DisplayMonitor")))

winreg.SetValueEx(self.pptregistry, "DisplayMonitor", 0, winreg.REG\_SZ, self.states["screens"][self.states["display"]["outputID"]]["physical"])

winreg.SetValueEx(self.pptregistry, "UseAutoMonSelection", 0, winreg.REG\_DWORD, 0)

self.PPTpresentation.SlideShowSettings.ShowPresenterView = False

self.PPTpresentation.SlideShowSettings.Run()

self.PPTpresentation.SlideShowWindow.View.AcceleratorsEnabled = False

self.overlay.setGeometry(self.screen)

self.overlay.showFullScreen()

[action(value) for value, action in reset]

except Exception as e:

print(e)

else:

*# Play with VLC*

self.player.set\_hwnd(int(self.foreground.winId()))

self.VLCmedia = self.vlc.media\_new(data["path"])

self.player.set\_media(self.VLCmedia)

def VLCposition(self):

"""

Get media progress (percentage)

"""

return self.player.get\_position()

def VLCpause(self):

"""

Pause the media

"""

if self.type != "media":

return

self.player.set\_pause(True)

self.VLCpaused\_c = True

def VLCplay(self):

"""

Play/Resume the media

"""

if self.type != "media":

return

self.player.play()

self.VLCpaused\_c = False

def VLCstop(self):

"""

Stop the media

"""

if self.type != "media":

return

self.player.stop()

def VLCmute(self, state):

"""

Mute audio

"""

if self.type != "media":

return

self.player.audio\_set\_mute(state)

self.VLCmuted\_c = state

def contentShow(self):

"""

Shows content on the output window

"""

if self.type == "media" and self.VLCpaused:

if not self.VLCmuted\_c: self.player.audio\_set\_mute(False)  
 *# Only unmute if the user did not force mute*

if not self.VLCpaused\_c: self.player.play()  
 *# Only resume if the user did not force pause*

self.VLCpaused = False

self.show()

self.foreground.show()

def contentHide(self):

"""

Hides content in the output window

"""

if self.type == "media":

if self.states["display"]["backgroundmedia"]:

self.player.set\_pause(True) *# Check background media behaviour*

elif self.states["display"]["backgroundaudio"]:

self.player.audio\_set\_mute(True) *# Check background media behaviour*

self.VLCpaused = True

self.foreground.hide()

self.show()

def contentDesktop(self):

"""

Reveals the desktop

"""

self.contentHide()

self.hide()