Qt 3D Studio — Digital cluster for a car — Part II

Oskar Lewandowski Follow
Feb 27, 2019 · 9 min read

In the last tutorial, we have created a working 3D presentation with navigation in the background. This time we will add other features like adding custom navigation style, DataInputs providing an interface to QML and extending our cluster to appear more realistic.

Since the last episode, a few aspects have changed. Even though I switched to Linux platform (Ubuntu 16.04), the implementation in Windows environment should work fine as well. Also, there was new Qt and Qt 3D Studio released. Remember to use MinGw compiler on Windows platform! It is required by MapboxGL plugin (https://doc.qt.io/qt-5/location-plugin-mapboxgl.html).

Going through this tutorial, you will be able to create an Instrument Cluster looking like the example below:

Requirements

You need following prerequisites to complete the tutorial:

- Qt 3D Studio 2.3 (but previous versions like 2.1/2.2 shall be good as well)
- Qt 5.12.1

- Qt Creator
- Mapbox plugin (it is provided with Qt 5.9+ www.mapbox.com/qt/)
- mapbox.access_token — generated after the creation of an account on www.mapbox.com

Preparation

Current 3D Presentation is based on the previous one from Part I but it is much more complex. Hence, I will focus on describing the features used to create it. There was a lot of tweaking and fighting with Qt 3D Studio 2.3 to make presentation work as expected. Qt3DS is a great tool, still, have some functionality gaps (I mention them later) but I hope they will be eliminated in the nearest future.

In case of any appearing problems, take a look at the **Troubleshooting**section at the end of the tutorial.

Prepare QML Project

Right before we start with **Qt Quick Application — Empty,** name your project**,** check Qt 5.12 and Finish. Before we dive into source code, we need to open the project file, in my case Qt3DStudio_Part2.pro and add the following modules:

QT += widgets qml quick 3dstudioruntime2 gui

We need, in particular, 3dstudioruntime2 in order to include Qt 3D Studio headers and libraries. Move to the source code.

In jump to main.cpp

- **Add header:** *#include <q3dsruntimeglobal.h>*

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <q3dsruntimeglobal.h>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);

    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

main.cpp hosted with ♥ by GitHub view raw

- Inside **qml.qrc** add three QML files named: Navigation.qml, Presentation3D.qml, Simulation.qml
- **Navigation.qml**

There have been small alterations done in comparison with the previous version. Imports use a newer version of Qt**Location** and QtPositioning.

- Added two properties:

```
property real carPositionX: 40.755
property real carPositionY: -73.995
```

Snippet.qml hosted with ♥ by GitHub view raw

“carPositionX/Y” variables define geographic start position and by incrementing or decrementing them, determine which direction and how fast position will be changed. “moveCamera” function is called from *navigtionLoop Timer.*

- Add function "***moveCamera***" to move the camera over the map.

To set a proper camera position, we need to assign our position to property provided by Map object:

```
center: QtPositioning.coordinate(carPositionX, carPositionY)
```

Snippet.qml hosted with ♥ by GitHub view raw

Coordinates are stored by **QtPositioning.coordinate** where the first argument is latitude and the second one longitude.

```
import QtQuick 2.12
import QtLocation 5.12
import QtPositioning 5.12

Item {
    id: naviFrame
    visible: true

    property real naviTilt: 30
    property real zoom: 16.0

    property real carPositionX: 40.755
    property real carPositionY: -73.995

    Simulation {
        id: simulation
    }

    Plugin {
        id: mapboxglPlugin
        name: "mapboxgl"

        PluginParameter {
            name: "mapboxgl.access_token"
            value: "pk.eyJ1Ijoib2xld2FuZG93c2tpMiIsImEiOiJjampqdnQ4MWIwdzlnM2ttZGVqcnYza2xvIn0.n
        }

        PluginParameter {
            name: "mapboxgl.mapping.cache.size"
            value: 2000
        }

        PluginParameter {
            name: "mapboxgl.mapping.additional_style_urls"
            //value: "mapbox://styles/olewandowski2/cjrakjdzm2jf12sp9rdgwncmh"
            //value: "mapbox://styles/olewandowski2/cjn5sohnz0nr42suv510nrw4l"
            value: "mapbox://styles/olewandowski2/cjrp872n3bo7i2slfyx8jlv75"
        }
    }

    Map {
        id: myMap
        anchors.fill: parent
        plugin: mapboxglPlugin
        center: QtPositioning.coordinate(carPositionX, carPositionY)
        zoomLevel: zoom
        bearing: 30
        tilt: naviTilt
    }

    function tiltIn()
    {
        if(naviTilt<160)
            naviTilt+=0.1
    }
    function zoomIn()
    {
        if(zoom < 17.0)
            zoom+=0.001
    }

    function moveCamera()
    {
        carPositionX += 0.000007
        carPositionY += 0.000005
    }

    Timer {
        id: navigtionLoop
        interval: 16
        repeat: true
        running: true
        triggeredOnStart: true
        onTriggered: {
            tiltIn()
            zoomIn()
            moveCamera()
        }
    }
}
```

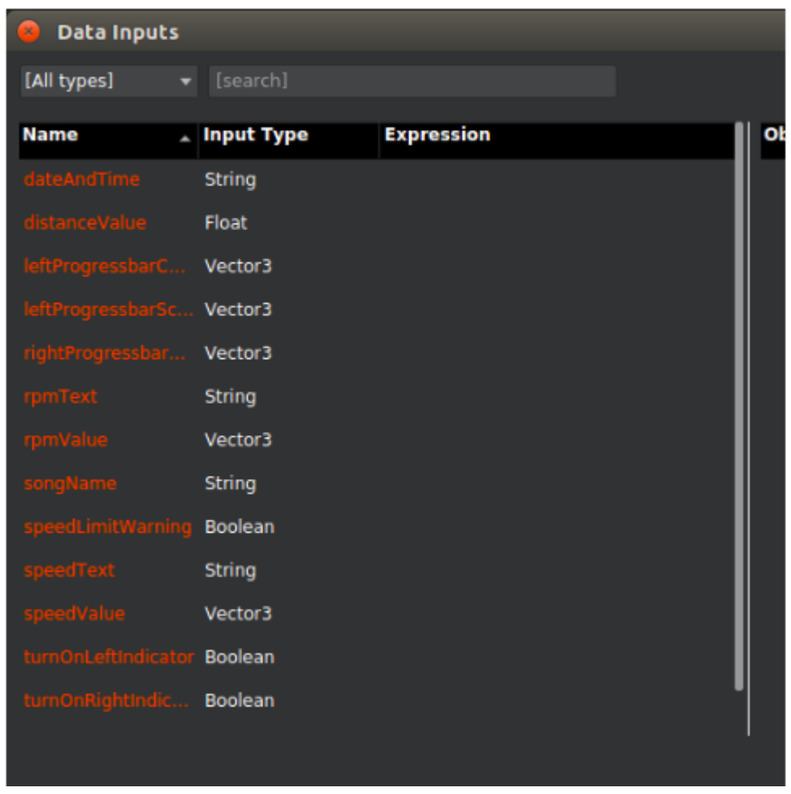
- **Presentation3D.qml**

Now, we use the newest snapshot version of QtStudio3D module 2.3 (fun fact: there is no 2.3 branch on the GIT repository). This time we want to push some data to the 3D presentation, thus it would be nice to have some simulation data, for instance, to simulate speed, rpm, turn signals, songs name, time and many others.

There is Simulation.qml already and it is going to be used to provide us with all the necessary data.

Let's add to Presentation3D.qml: **Simulation { id: *simulation* }**

I assume we're having Qt 3D presentation which contains dataInputs as below (In Qt3DStudio go to File→Data Inputs…):



The nice aspect is that we can map them to QML variables and expose data.

In this case inside Presentation { } block add type:

```
DataInput {
  name: "speedValue"
  value: simulation.inputSpeedValue
}
```

Snippet.qml hosted with ♥ by GitHub view raw

name: *speedValue* — used in Studio to control a rotation of speed needle

value: maps property *inputSpeedValue* from Simulation object. Thus, if inputSpeedValue changes, the needle object inside presentation rotates.

Using dataInput in Studio seems to be simple. In order to do this, select an object you want to control, click an icon next to property (orange "three-dots") and select variable from popup:

Find more information here: https://doc.qt.io/qt3dstudio/using-data-inputs.html

For some reason, data types in Qt 3D Studio are named a bit differently than in the QML. For example, Bool vs Boolean, Real vs Float.

Properties like rotation, position or scale are 3D vectors and that is why we need to map them to vector3d type.

Speed needle is controlled from Simulation and is declared this way:

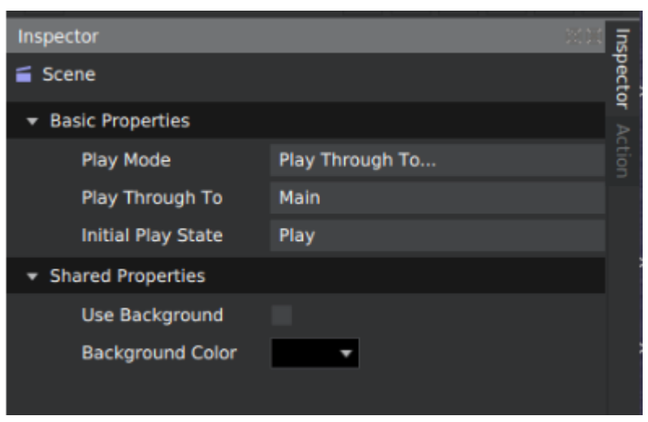
```
property vector3d inputSpeedValue: Qt.vector3d(0, 0, 120)
```

Now we know how dataInput works and we can map other inputs to in QML (check Presentation3D.qml for details).

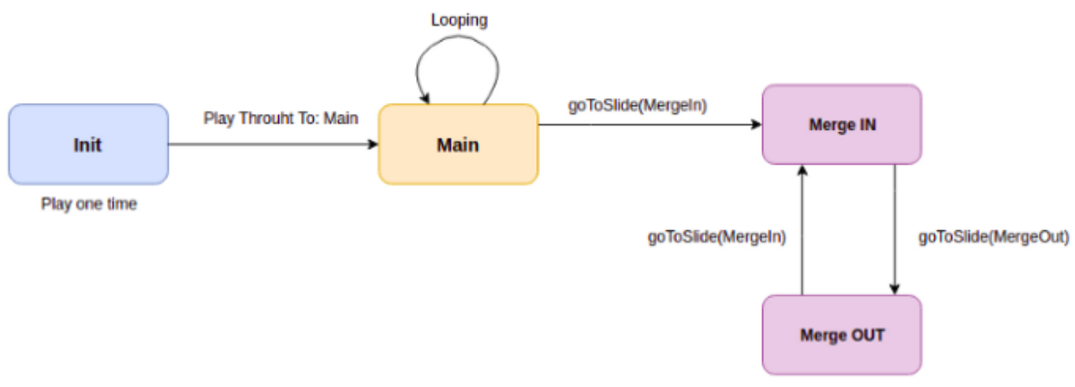
The last thing in Presentation 3D is a Timer which switches the presentation slide every 10 seconds. It shows how goToSLide function can be used and how we can manipulate slides using names.

```
s3d.presentation.goToSlide("Scene", switchSlide)
```

In this tutorial, I used two solutions to switch slide. The first approach is to set up each Slide → Scene properties:



“Init” slide after playing is going to next slide named “Main”. Another way to switch slide is to use goToSLide function from QML code. State machine of slides works as below:



Perhaps, it is not the best solution, but it works as planned. Probably using sub-presentation in some cases would be more efficient.

- **Simulation.qml**

Responsible for generating data to simulate engine behaviour (rpm, speed, gears, oil temperature, fuel), turn indicators, time and other data.

I did not focus on the accuracy of the generated data. It is only used to show how can we pass information via dataInputs to the 3D presentation.

It contains 3 loops: simulationLoop, lights, infotainment. Each one behaves as an infinite loop with a different time interval.

Review code on your own, it is not complicated but there is no reason to dive into details.

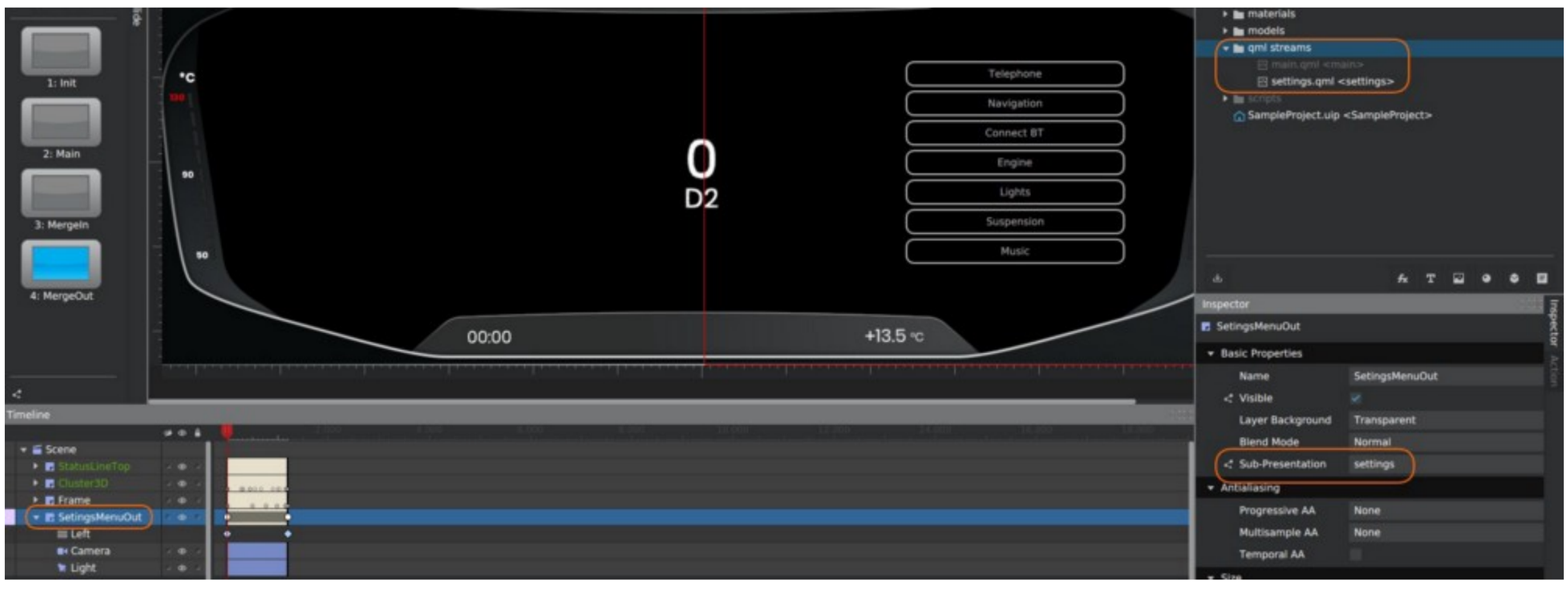
Also, you can implement your own data simulator.

- **settings.qml**

That file is located in **Resources/digitalcluster.qrc/Cluster/qml streams/**

```qml
import QtQuick 2.2
import QtQuick.Controls 1.2


Image {
    width: 640
    height: 480

    Rectangle {
        id: settingsScreen
        width: 640; height: 480
        opacity: 0.9
        color: "black"

        Column {
            anchors.centerIn: settingsScreen
            spacing: 5
            Repeater {
                model: ["Telephone", "Navigation", "Connect BT", "Engine", "Lights", "Suspension

                Rectangle {
                    x: 50; y: 0
                    width: 260; height: 30
                    border.width: 2
                    border.color: "white"
                    color: "black"
                    radius: 10
                    opacity: 0.8

                    Text { anchors.centerIn: parent; color: "white"; text: modelData }
                }
            }
        }
    }
}
```

It is possible to take QML file and use it as a sub-presentation in Scene layer



It is a nice feature very likely to use in developer- designer cooperation. The former one can prepare a ready solution, so the latter one uses it as an element that will be adjusted on the Scene. In same way should be possible to embed mini navigation between gauges or to display music album streamed from QML.

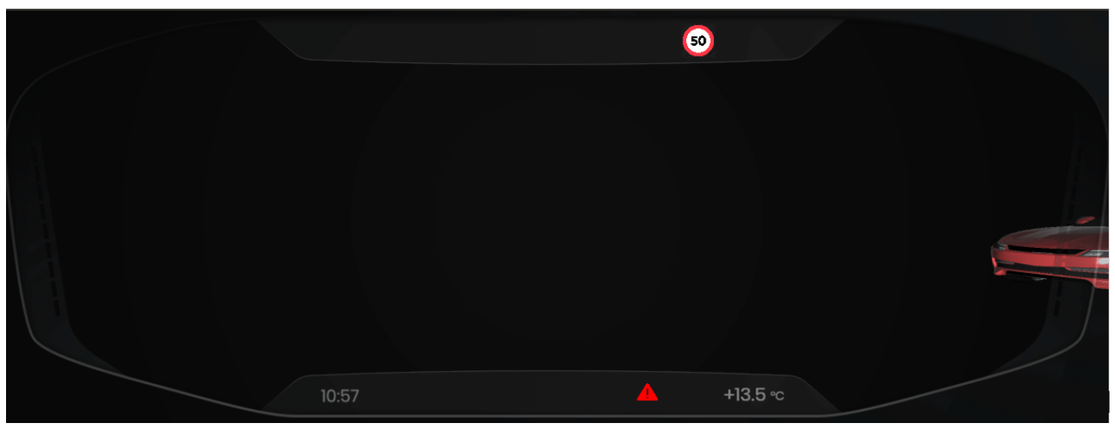
But not everything works perfectly yet. I found out that compilation in Release mode creates a problem with parsing metadata from QML streams. After a short research, I created an issue ticket on Qt 3D Studio https://bugreports.qt.io/browse/QT3DS-2967) and move on.

Working with Qt 3D Studio 2.3

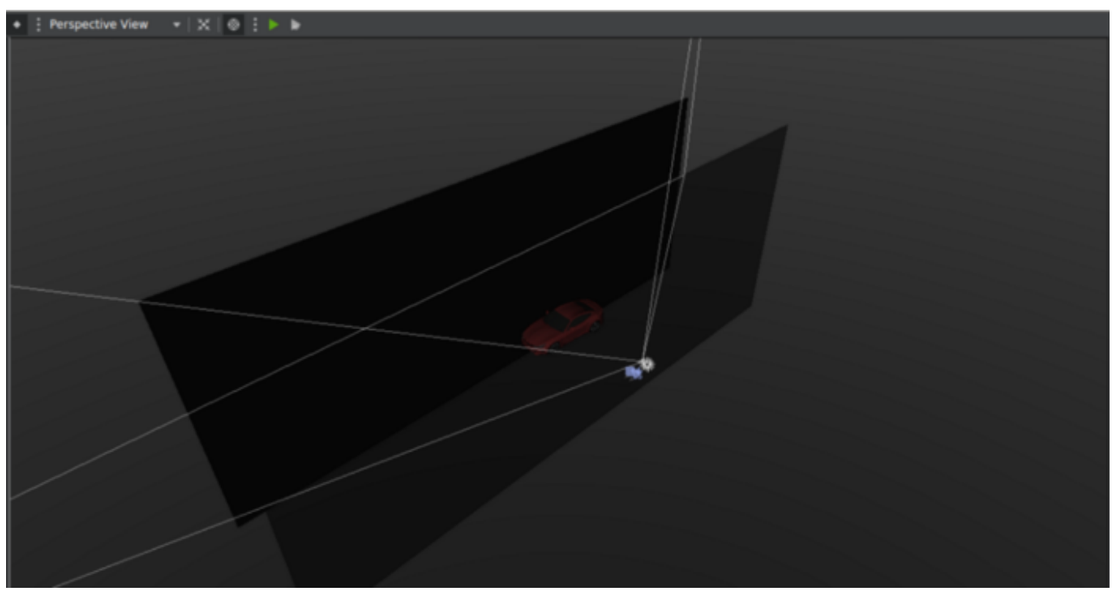
I usually waste a lot of time to tame animations and look for 3D elements. The main visible problem refers to the preview window which gives different visual result than the one produced in our application. Lights behave differently, like it was too strong. Another thing is animating opacity property on ring gauges. The result was hiding 3D element permanently or in halfway behaved like switching visible property on/off where I expected smooth fade out/fade in effect. In some places, I had to skip some solutions or replace them with a trick. Once I open project on Windows platform and presentation looks different than the one I made on Ubuntu.

Without snapping option 3D Studio forced me to become a sniper while working on a time-line. It would be nice to have to a possibility to turn on/off such function.

For example, there was a problem with opacity applied directly on 3D car model (some parts smoothly disappeared and other just toggle straight from 100% opacity to 0%). The model presented below, shall go from 100% opacity to 0% and after 1s go back to 100% while spinning.



I solved it with a trick:



The model was placed between two planes, one is a static background and the second one placed between a camera and a 3D model. When opacity is changed on the foreground plane, it works as fade in/out effect.

On the bright side, it is possible to turn on profiling window which provides a lot of useful information:

To turn it on just add Studio3DProfiler inside Studio3D section, like below:

```
Studio3D {
        Studio3DProfiler {
                id: profiler3d
                visible: true
                anchors.fill: parent
                anchors.topMargin: 40
        }
        Studio3DProfiler {
            id: profiler3d
            visible: true
            anchors.fill: parent
            anchors.topMargin: 40
        }
}
```

Snippet.qml hosted with ♥ by GitHub view raw

Source code available:
https://www.dropbox.com/s/66apgiw8qzdk8r1/Qt3DStudio_Part2.7z

Final effect:

Network error

A network hiccup interrupted playback. Please reload the player and tr

https://vimeo.com/319007088

Summary

Generally, Qt 3D Studio is a very promising tool that I hope will be a good alternative for Kanzi or CGI Studio. But there is still a lot of work to do.

Please do not focus on things like turn indicator timings which are not realistic. This was not a point of this tutorial. But I encourage you to extend this presentation in terms of modifying it in a way you like and to share the results in the comment section.

Things that can be done:

- create components

- add interactive buttons: next/prev song, turn right/left indicator

- manipulating a colour of the progress bar (make id red when low fuel or engine temperature is too high).

- working settings menu, maybe to control which mapbox styling is active? (hint: check how property activeMapType and supportedMapTypes works)

At this moment, this will be the last part for Qt 3D Studio until bigger changes will be introduced. Next topic will be related with Kuesa module for Qt 3D. I have seen their presentation at QTWS at Berlin and I was impressed how good it was.

Useful links

Documentation:

https://doc.qt.io/qt3dstudio/

https://doc-snapshots.qt.io/qt3dstudio/runtime/

Repositories:

https://codereview.qt-project.org/#/admin/projects/qt3dstudio/qt3dstudio

https://codereview.qt-project.org/#/admin/projects/qt3dstudio/qt3d-runtime

KDAB:

https://github.com/KDAB/kuesa

Other:

https://www.mapbox.com/qt/

Troubleshooting

1. Maybe it was only me but there was a problem with running 3D presentation and maboxgl at the same time. I found in logs line like this:

Actual format is ***QSurfaceFormat****(version 4.5, options QFlags<QSurfaceFormat::FormatOption>(), depthBufferSize 0, redBufferSize 8, greenBufferSize 8, blueBufferSize 8, alphaBufferSize 0, stencilBufferSize 0, samples -1, swapBehavior QSurfaceFormat::DefaultSwapBehavior, swapInterval 1, colorSpace QSurfaceFormat::DefaultColorSpace, profile QSurfaceFormat::CoreProfile)*

renderer: Mesa DRI ***Intel(R) HD Graphics 630*** *(Kaby Lake GT2)*

Ok then, my laptop has two graphics cards: Intel and GeForce. Also, there was a log with information about unsupported framebuffer format:

Framebuffer blits are not supported by ES 2.0 (since ES 3.1)

Clearly integrated Intel GPU do not support OpenGL ES 3.1 version. In my case there was no dedicated dgpu driver for nVidia gpu, I downloaded one from the website and installed. It wasn't enough, though. Remember that secure boot option in BIOS can block access to nvidia gpu so it should be set as disabled. Then try to run an installed program with drivers. It's called "NVIDIA X Server Settings"

2. Terminating app due to uncaught exception 'mapbox::sqlite::Exception', reason: 'database or disk is full'

After a week of work with mapbox it started throwing the exception like above. There is a ticket for this issue (https://github.com/mapbox/mapbox-gl-native/issues/7164) but it looks like it might still occur.

I did not found a solution for this problem since I switched on the second machine with ubuntu and changed mapbox cache size to 2000MiB (just because I can). Until now everything works fine.

PluginParameter { name: "mapboxgl.mapping.cache.size" value: 2000 }

Qml Qt Automotive Software Development Qml Development