Project Work 2

Video Viewer For Eye-tracking Platform

Degree course:	Electrical and Communication Engineering
Authors:	Timothée Mollet
Tutor:	Prof. Dr. Theo Kluter
Date:	06.06.2016

Versions

Version	Date	Status	Remarks
0.1	16.05.2016	Draft	Create Document
0.2	06.06.2016	Draft	First Complete Draft

Management Summary

Eye tracing has a wide area of applications such as sports and psychology. To analyze gaze direction from a eye-tracking system for sports, software is needed that can display the gaze position on a video playback. Each gaze point needs to be synchronized with the video.

The target for this project was to develop such a software that can be expanded later but is feature complete so that it can be used for analyzing footage and gaze data.

Contents

1. Introduction

1.1. Analyzing Eye Tracking

Eye tracking is used to measure where someone is looking. It is used in a wide variety of applications such as marketing research, psychology, virtual reality and sports training.

To enable high speed eye tracking for sports the HuCE developed an eye tracking system called the Gazelle Eye Tracker that is fast, portable and built for outdoor usage.

For analyzing the recorded footage a player is required that can put an overlay over the video playback. This overlay needs to be synchronized with each frame of the video.

1.2. Requirements

There a few key requirements that can be categorized in optional and must have. This is done in 1.1.

Requirement	must	optional
Play video	x	
Video has overlay	x	
Step frame for frame forward and backward	x	
Step overlay for overlay forward and backward	x	
Overlay and Frames are in sync	x	
Display data of eye-cameras		x
Play at various playspeeds		x
Play overlays		x

Table 1.1.: List of requirements

2. Software Design

The Software is developed with the Qt-Creator and uses Qt-5.6 and OpenCV 3.1 as libraries. Coding convention is documented in the Appendix. Additionally Doxygen is used to document the code.

2.1. Design Decisions

Choosing the right container for each dataset has major impact on resource usage and performance. So they need to be evaluated carefully.

2.1.1. Storage Containers

Qt has an implementations for all popular storage containers. They all have different structures and are intended for different applications. The Containers can be categorized in two major categories. The first are sequential container each element is stored at a certain index and can be accessed over said index. The second are associative containers and set where all elements are stored with a key. The algorithmic complexity of the containers are shown in Table 2.1 for the sequential containers and in Table 2.2. O(n) stands for linear time so it scales linearly with the size of a container. O(1) stands for constant time, so each operation takes a fixed amount of time no matter how big the container is. O(log(n)) equals linear time. So the cost of an operation on a container raises logarithmic to the number of items in that container. If the behavior is not guarantied it is specified as amortized.

	Index lookup	Insertion	Prepending	Appending
QLinkedList<T>	O(n)	O(1)	O(1)	O(1)
QList<T>	O(1)	O(n)	Amort. O(1)	Amort. O(1)
QVector<T>	O(1)	O(n)	O(n)	Amort. O(1)

Table 2.1.: Algorithmic Complexity of the sequential Qt Containers where each element can be accessed over an index [1]

	Key lookup		Insertion	
	Average	Worst Case	Average	Worst case
QMap<Key, T>	O(log n)	O(log n)	O(log n)	O(log n)
QMultiMap<Key, T>	O(log n)	O(log n)	O(log n)	O(log n)
QHash<Key, T>	Amort. O(1)	O(n)	Amort. O(1)	O(n)
QSet<Key>	Amort. O(1)	O(n)	Amort. O(1)	O(n)

Table 2.2.: Algorithmic Complexity of the associative Qt Containers where each element can be accessed over the associated key [1]

QLinkedList

QLinkedList stores elements by pointing to the previous and next elements, creating a chain that is linked together. Because the pointer to an element is only stored in the adjacent elements, it is slow when accessing an element over an index. However when accessed over an iterator it is possible to insert an element at any point during the traversal of the linked list.[3]

QList

QList allows for fast access over an index and allocates space before and after its internal array which usually results in constant time insertion on both ends of the list as shown in 2.1. It has to be noted that QList allocates its elements on the heap when they use more storage space than a pointer.[4]

QVector

QVector is similar to QList as it can also be accessed over an index in constant time. Because QVector doesn't allocate storage before the array it is not possible to prepend objects in constant time. QVector stores each element successively.[8]

QHash

QHash uses a hash table to store its associated key-value pairs. The advantage of that is fast insertion and lookup of a key, usually in constant time. The hash table is not sorted, so an item with the key "15" can be followed by an item with the key "4". QHash automaticly grows and shrinks according to the number of stored elements. [2]

QMap

QMap is similar in the usage as a QHash. The main difference is that QMap sorts the keys. So an item with the key "4" is followed by an item with the key "15" when there is no key between the two values. The trade-off for that is slower lookup and insertion of logarithmic time. [5]

QMultiMap

QMultiMap has the same properties as a QMap but is more convenient if you want to store multiple values per key.[6]

QSet

QSet is similar to QHash but it does not store a value. An application for a QSet is to store a stream of values without storing duplicates. [7]

2.1.2. Containers in Gazelle View

In Gazelle View there are multiple sets of data that needs to be stored. There are the decoded Frames that need to be handled dynamicaly, the position an the timestamp of each overlay and the timestamp for each frame. Each of those datasets has different requitement and the best storage container needs to be evaluated.

Framebuffer

Each frame contains a large amount of Data. Around 6 MB per picture for a fullHD stream. Additionally it is also important to store the associated framenumber for each frame. As the buffer is usually a continuous flow of data this can be done with a sequential container with an offset or an associative container with the key represented as frame. Disadvantage of sequential containers are that if you move the first stored frame to the first index the container needs to be rearranged each time frames are deleted. If the frames are stored at the index of their framenumber you have a very large container with mostly empty elements and you need to keep track where how many elements are stored to free the buffer efficiently.

Because the frames don't need to be sorted it is best to use a QHash to store a frame with the framenumber as key. It allows for uncomplicated insertion and checks of how many frames are already stored for handling of the buffersize.

Overlays

An overlay consists of the position where it should be placed and the timestamp when it should be displayed. Access is usually over a timestamp to get an overlay that is before or after said timestamp. To accomplish that the container needs to be associative and sorted. QMap fulfills those requirements perfectly.

Timestamps

The timestamps consists out of a continuous list of timestamps for each frame. It needs to be possible to access the timestamp for each frame and the frame before a timestamp. There are a few options to solve this dual access problem.

The first is a QVector where each timestamp is stored at the index of the corresponding frame. The advantage of that approach is that it is not memory intensive and access of a timestamp of a frame is done in constant time. But o get the frame for a timestamp requires iterating over the QVector resulting in linear time access. As there may be multiple of 10'000 frames per video this is not desirable.

Another option would consist of a QVector for the timestamps and a QMap for accessing the framenumber from a timestamp. The access over a QMap reduced the time needed from linear to logarithmic time for the cost of duplicating the stored information.

A third option exploits the fact that timestamps count up. So it is possible to get the correct frame out of timestamps stored in a QVector in logarithmic time with the usage of successive approximation to calculate the frame. As this combines the small memory footprint as the first and fast access of the second option it is the best choice for this problem.

3. Gazelle View

Gazelle View is the software developed in this project.

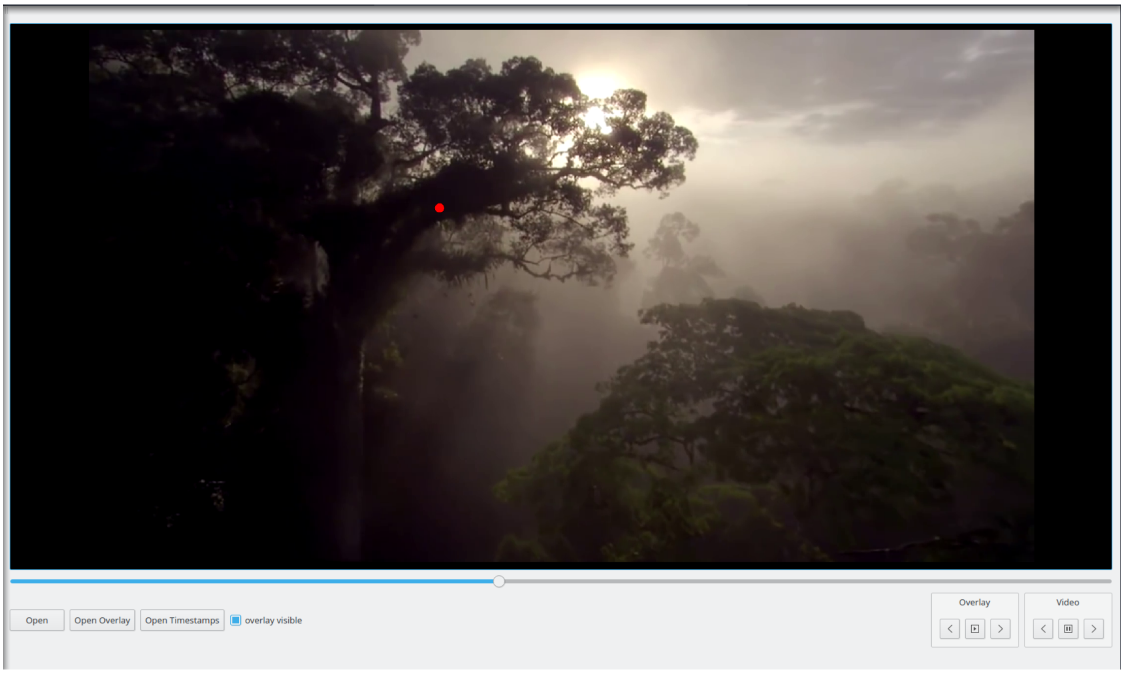


Figure 3.1.: GUI of Gazelle View with the Breeze icon theme

3.1. Use cases

The use cases for Gazelle View are presented in 3.2 and are deducted from the requirements.

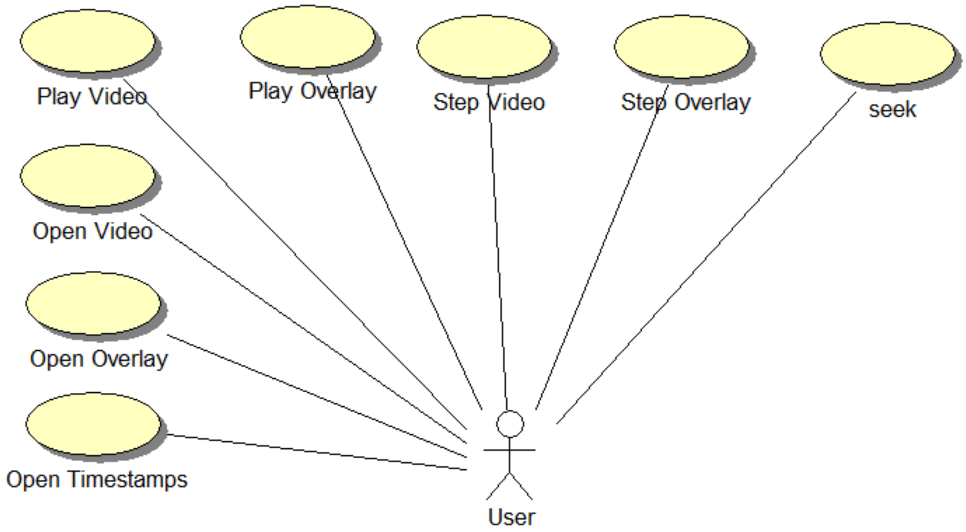


Figure 3.2.: Use Cases for Gazelle View

- The user can open a video, overlays and timestamps
- The user can play the video
- The user can pause the video
- The user can go to the next or previous frame
- The user can seek
- The user can play overlays
- The user can pause overlays
- The user can go to the next or previous overlay

3.2. Class Diagram

Gazelle View is build with four classes as seen in 3.3 The GUI is handled by the class MainWindow. User input is forwarded to the class VideoHandler. VideoHandler is in control of which overlay and frame are displayed or decoded and it also manages the buffer. Because decoding is an expensive operation, so VideoHandler orders frames to be decoded from DecodeWorker. DecodeWorker has as sole instance access to the videostream and has to provide metadata such as framerate and how total count of frames in the stream. DecodeWorker runs in it's own thread so it doesn't block the other classes. The class Overlays is responsible to parse overlays and timestamps and find the corresponding frame or overlay for a timestamp.

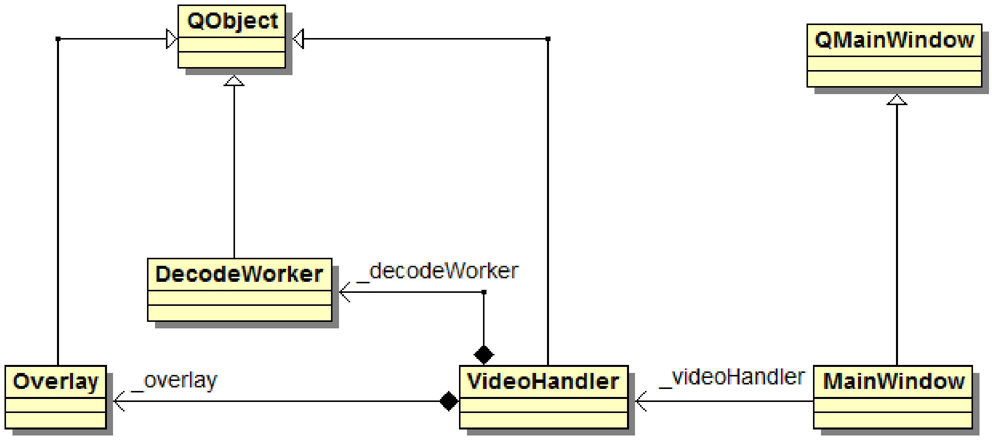


Figure 3.3.: Class Diagram for Gazelle View

3.2.1. MainWindow

MainWindow handles the GUI, user input and displays the image with the overlay. It extends from QMainWindow and allows using the signal/slot framework from Qt. MainWindow sets up the icons of the play/pause, back and forwards buttons for overlay and video so that they use the icon theme of the current desktop environment. Fallback icons are provided for system that don't have themes or themes that don't have the necessary icons. An example of the GUI with the Breeze icon theme from KDE can be seen in 3.1.

The communication with the class VideoHandler is done over the signal/slot framework from qt. The slot "displayImage" gets called when VideoHandler want's to display a new image. This image is shown on a QGraphicsScene with black background.

3.2.2. VideoHandler

VideoHandler handles a few things.

- Order new frames to be decoded
- Send a frame to be displayed
- Handle the framebuffer
- Keep overlays and Frames in sync
- Send overlays to be displayed

VideoHandler extends QObject to communicate with the other classes over the signal/slot framework.

VideoHandler relies on DecodeWorker to decode the frames in a separate thread and on Overlays for getting the information which overlay needs to be displayed on which frame.

Framebuffer

As decided in 2.1.2 this is done with a QHash with the framenumber as key and a pointer to the Image as value. The buffer is trailing so it stores the last 50-100 frames. This is because the stream can only be decoded forward and jumping to a previous frame is expensive. To keep the size of the framebuffer reasonable a method iterates over the QHash when it get's to large and deletes all frames that are more than 50 frames behind.

3.2.3. Overlay

The class Overlays is responsible for the following things.

- parse file with information about overlays
- parse file of timestamps for the frames
- store overlays and timestamps
- return the overlay before or after a timestamp
- return the frame before a timestamp
- return the timestamp for a certain frame

The unit of the timestamps don't matter as long as they are the same for overlays and frames. As discussed in 2.1.2 a QMap stores the information for the overlays. The next or previous overlay is accessible for a timestamp. Additionally the corresponding timestamp for the overlay is also returned.

Timestamps for Frames

It is required to provide access for both the timestamp for a certain frame and the frame for a certain timestamp. In 2.1.2 successive approximation was proposed to solve this issue. The implementation is as follows:

```
for (int i = size/2; i >= 1; i /= 2) {
    if ((frameCount > (currentFrame + i)) &&
            (_sceneFrames.at(currentFrame + i) < timestamp)) {
        currentFrame += i;
    }
}
```

- `_sceneFrames` is the QVector that stores timestamps
- `frameCount` is the size of `_sceneFrames`
- `size` is the next higher power of two after `frameCount`
- `currentFrame` holds the result at the end of the algorithm
- `currentFrame + i` is the frameindex that is tested

The result converges to the frameindex that has the timestamp below the timestamp that is supplied. It does so by comparing the supplied timestamp with the timestamp at index $\frac{\texttt{size}}{2}$ of `_sceneFrames` first and followed by either $\frac{\texttt{size}}{4}$ or $\frac{3 \text{ x } \texttt{size}}{4}$ depending on the outcome of the comparison. This goes on until the increment is one.

3.2.4. DecodeWorker

The job of the DecodeWorker is to decode frames and convert them into a usable format so that the frame can be displayed on screen on a QGraphicScene. A library that enables decoding frames and give access to the pixeldata of each frame is OpenCV. It would also be possible to play videos with QMultiMedia. The disadvantage of this is that there is no direct access to each frame and the pixel data. OpenCV was chosen because i in the future it might be a requirement to manipulate or analyze each frame of the stream and OpenCV provides many tools that enable that.

Decoding and converting a video stream takes time so DecodeWorker is moved to it's own thread so it doesn't block the GUI or other parts of the program. Resulting race conditions are mitigated by the use of QMutex.

Getting a Frame Ready

The frame needs to be stored as a QPixMapItem in order to be displayed on a QGraphicsScene. This is done by a few steps:

1. Decode a single frame with OpenCV into a Mat
2. Convert the Mat from BGR to RGB
3. Transfer the picture data from the Mat to a QImage
4. convert the QImage to a QPixMap

The second step is required because OpenCV stores its images in the BGR format instead of the more common RGB format.

4. Performance

Showing decoded pictures on the screen takes some time. The scene camera of the eye-tracking system record with 30 frames per second at a resolution of 1920x1080. Translated into time that means for each frame there are on average $\frac{1\text{s}}{30} = 33.3ms$ time for it to be decoded and converted so it is ready to be displayed.

4.1. Measurement

The best way to determine the performance of something is to measure it.

4.1.1. Test Setup

The Measurement was done on a ASUS Zenbook uv32vd with the following specifications:

CPU	Intel Core i7-3517U
Memory	10Gb DDR3
SSD	Samsung SSD 830 Series (256GB)
OS	Arch Linux x86_64
Qt	5.6.0-7
OpenCV	3.1.0-3

For each frame the following durations are recorded:

- Decode from the videostream
- Conversion from BGR to RGB
- Data Transfer to QImage
- Conversion to QPixmap

Each duration is measured with QTime and written to a logfile. The measurement is done with a Release version of the GazelleView. The Video used is "Planet Earth: Amazing nature scenery (1080p HD)":

`https://youtu.be/6v2L2UGZJAM`

4.1.2. Results

The CPU hovered between 2.6 and 2.7 GHz during the playback and the utilization was between 40 and 45%. Ram usage of Gazelle View stayed around 1Gb. In total 24228 frames where measured not including the first frame.

The average duration each step took is listed in 4.1. With a total average of 16.5ms it should not be a problem to decode and convert each frame in time. But because there are frames that take longer than 33ms it is better to decode 2-3 frames before the current frame so a slow frame won't cause stutter. Not surprising is that decoding is the most expensive operation followed by the conversion from BGR to RGB.

	Average	Maximum
Decode	7.8 ms	28 ms
BGR to RGB	5.0 ms	17 ms
To QImage	0.0 ms	2 ms
To QPixMap	3.7 ms	12 ms
Total	16.5 ms	36 ms

Table 4.1.: Average and Maximum time for each measured step

5. Conclusion / Results

The Software that was developed during this project fulfills all must requirements. Additionally the functionality to play overlays was also implemented and the performance of the video decoding and converting was evaluated.

With the developed software it is now possible to analyze where someone is looking if all needed information is available.

There was not much time to test the software extensively so it probably has a fair amount of bugs. GazelleView is still missing many features that a up to day video player has such as fullscreen or audio support.

Declaration of primary authorship

I hereby confirm that I have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Biel, 06.06.2016
Last Name, First Name:	Timothée Mollet
Signature:	....

Glossary

BGR BGR is similar to RGB encoding but with red and blue swapped.

GUI Graphical User Interface.

KDE KDE is a free software community that produces software such as the Plasma Desktop and the KDE Frameworks.

RGB RGB is a color model where red, green and blue are added together.

Bibliography

[1] T. Q. Company, “Qt documentation: Containers,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/containers.html

[2] ——, “Qt documentation: Qhash,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qhash.html

[3] ——, “Qt documentation: Qlinkedlist,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qlinkedlist.html

[4] ——, “Qt documentation: Qlist,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qlist.html

[5] ——, “Qt documentation: Qmap,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qmap.html

[6] ——, “Qt documentation: Qmultimap,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qmultimap.html

[7] ——, “Qt documentation: Qset,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qset.html

[8] ——, “Qt documentation: Qvector,” accessed: 2016-05-16. [Online]. Available: http://doc.qt.io/qt-5/qvector.html

List of Figures

List of Tables

APPENDICES

A. Coding Guidelines

The Coding Guidelines are not written by me, they are a copy of our coding guidelines at work at Ruag. I got permission to include them in this document.

A.1. Coding Style

- In general qtcreator >= 3.3 default C++ (Qt built-in) code style
  - `http://wiki.qt.io/Coding-Conventions`
  - `http://wiki.qt.io/API_Design_Principles`
- Class names: Start with capital letter, continue with Capital Letter for each word. SourceFilter.
- File Names: source_filter.h source_filter_p.h source_filter.cpp (as it is)
- Use **"abstract"** instead of "interface": AbstractDataSource
- Local Variables, **camelCase**: isEnabled
- Global Variables, prefix **with underscore**: _isEnabled
- **No** variable **prefixes** like ~~lisWidgets~~
- Function Names, camelCase: setEnabled
- Indention: **4 spaces** instead of tab
- Doxygen: Class, Public Functions, Enums and Private Functions have a **doxygen comment in the header file**. Variables optional
- **Getter / Setter** should stick together in .h and .cpp
- Pointer symbol and ampersand belong to variable type (Foo* foo, Bar& bar)
- **Signal / Slots** have no prefix ~~slotStartExport~~
- Try to have a **minimal list of includes**.
- Your Code shall produce **no compiler Warnings!**

If/else statement

```
if (condition) {

} else {

}
```

Switch case

```
switch (control) {
case value:

    break;
default:
    break;
}
```

For loop

```
for (unsigned var = 0; var < total; var++) {

}
```

Range based for loop

```
for (auto const& e : container) {

}
```

While loop

```
while (condition) {

}
```

const / ampersand placement

```
void Foo::printDebug(QString const& text)
{
    qDebug() << text;
}
```

Connect

```
connect(instance, &Class::cameraChanged, ui->cameraBox, &QComboBox::setCurrentIndex);

connect(_controller, &Controller::updateViewport ,this,
    (void(QOpenGLWidget::*)())&QOpenGLWidget::update);

connect(ui->counterButton, &QAbstractButton::pressed, [=] () {
    int const currentValue = ui->counterButton->text().toInt();
    ui->counterButton->setText(QString::number(currentValue + 1));
});
```

A.2. Best Practices & Tips

QSharedPointer Illustration of a Common Problem:

```
int a;
QSharedPointer<int> pa (&a);
QSharedPointer<int> pb (&a);
pa!=pb //Both shared pointers will have a separate refcounting
```

You shall not pass a raw pointer to One Object more than once to a QSharedPointer (Constructor). If you have a second location where you need a QSharedPointer to the same Object then you have to Construct that SharedPointer with the Copy-Constructor.

Also: QSharedPointer<...>(this) is always wrong. Use: `http://doc.qt.io/qt-5/qenablesharedfromthis.html`

Useful Debugging Helper: `http://doc.qt.io/qt-5/qsharedpointer.html#optional-pointer-tracking`

QList and QVector

- **Do not use 'QList's.**

  They're officialy slow! 'QVarLengthArray', 'QVector', 'std::vector' and 'std::deque' are your friends. Why? QLists stores only pointer to elements, unless sizeof(T) < sizeof(void*) and you declared T to be a Q_MOVABLE_TYPE (with Q_DECLARE_TYPEINFO). More infos in the Slides from Oliver Gaffort `http://t.co/MEjWZhZ5L2`

- **Do not pass QLists and QVectors around as Pointers**

  QList and QVector are implicitly shared. Passing them around as pointers only raises questions about the ownership, and does not improve performance in any Way.

- **Do not use range based for loops on Qt-Containers without paying extrem attention!**

  Using a range based for loop on a Qt-Container which is non-const will cause a deep copy to occour even if you declare the element to be constant. (Reason: QVector::begin() causes a detach). Use 'foreach' (='QT_FOREACH') on qt containers and range-based for loops on std containers.

Custom (Pointer) Types in QVariant

If you have to store custom types in QVariant, store it with the correct type:

```
Bla* foo = ...;

//Wrong!!!!
QVariant v = QVariant::fromValue(static_cast<void*>(foo));
Bla* bar = static_cast<Bla*>(v.value<void*>());

//Correct:
QVariant v = QVariant::fromValue(foo);
Bla* bar = v.value<Bla*>();
```

But this means that I have to register T*_ with as a Metatype?!

Yes, in the most cases. But note that you don't have to use qRegisterMetaType<T*>(); because Q_DECLARE_METATYPE(T*) is enough.

Don't forget to free instances hidden in a QVariant

Common problem:

```
QVariant v = QVariant::fromValue(new Foo());
//...
//Programm terminates
//Value stored in Variant was never freed -> MemoryLeak.
```

Either ensure that the Owner of the Object deletes it, or store a QSharedPointer<Foo> in the QVariant!

B. Doxygen

Gazelle View

Generated by Doxygen 1.8.11

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

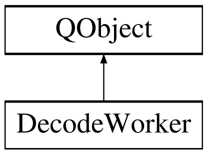
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 DecodeWorker Class Reference

Inheritance diagram for DecodeWorker:



Public Slots

- void decode (int frames)

  DecodeWorker::decode slot that decodes frames and converts them into a QGraphicsPixmapItem.

Signals

- void **resultReady** (QGraphicsPixmapItem ∗image, int frame)

Public Member Functions

- double **posMSec** () const
- double **posFrames** () const
- void setPosFrames (double frame)

  DecodeWorker::setPosFrames sets the Current Frame of the decoder to frame.
- double **posRelative** () const
- double **framerate** () const
- double **codec** () const
- double **frameCount** () const
- bool openStream (QString stream)

  DecodeWorker::openStream opens stream.
- void updateAllProperties ()

  DecodeWorker::updateAllProperties updated every property that can be accessed.

3.1.1 Member Function Documentation

3.1.1.1 void DecodeWorker::decode (int *frames*) [slot]

DecodeWorker::decode slot that decodes frames and converts them into a QGraphicsPixmapItem.

Parameters

frames	

3.1.1.2 bool DecodeWorker::openStream (QString *stream*)

DecodeWorker::openStream opens stream.

Parameters

stream	filename of the stream

Returns

true when successfull

3.1.1.3 void DecodeWorker::setPosFrames (double *frame*)

DecodeWorker::setPosFrames sets the Current Frame of the decoder to frame.

Parameters

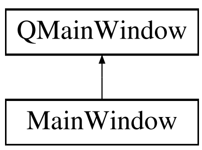
frame	

The documentation for this class was generated from the following files:

- decode_worker.h
- decode_worker.cpp

3.2 MainWindow Class Reference

Inheritance diagram for MainWindow:



Signals

- void freedImages ()

  freedImages The scene no longer holds ownership of any image

Public Member Functions

- **MainWindow** (QWidget ∗parent=0)

The documentation for this class was generated from the following files:

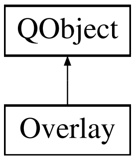
- mainwindow.h
- mainwindow.cpp

3.3 Overlay Class Reference

The Overlay class parses Overlays and Timestamps and gives access to them.

```
#include <overlay.h>
```

Inheritance diagram for Overlay:



Public Types

- enum ParserError { NoError, OpenFileError, ParseError }

  Overlay::ParseError Enum for storing euccess of parsing.

Public Member Functions

- QPair< QVector2D, qint64 > nextOverlay (quint32 timeStamp)

  Overlay::nextOverlay gets the next overlay after timeStamp.
- QPair< QVector2D, qint64 > nextOverlay ()

  Overlay::nextOverlay gets the next overlay after a timeStamp that is internaly stored.
- QPair< QVector2D, qint64 > previousOverlay (quint32 timeStamp)

  Overlay::previousOverlay gets the previous overlay before timeStamp.
- QPair< QVector2D, qint64 > previousOverlay ()

  Overlay::previousOverlay gets the previous overlay before a timeStamp that is internaly stored.
- QVector< QVector2D > overlaysToFrame (int frame)

  Overlay::overlaysToFrame.
- Overlay::ParserError parseOverlays (QString fileName)

  Overlay::parseOverlays parses the content of fileName as overlays. Each line that has more or equal to 3 tab separated values and doesn start with '#' is parsed.
- Overlay::ParserError parseFrames (QString fileName)

  Overlay::parseFrames parses the content of fileName as timestamp. Each line that has more or equal to 1 tab separated values and doesn start with '#' is parsed.
- qint64 timeStamp (int frame)

  Overlay::timeStamp returns the timestamp to frame in constant time.
- int frame (quint32 timestamp)

  Overlay::frame returns the next lower framenumber that comes before timestamp This is done in logarithmic time.

3.3.1 Detailed Description

The Overlay class parses Overlays and Timestamps and gives access to them.

3.3.2 Member Enumeration Documentation

3.3.2.1 enum Overlay::ParserError

Overlay::ParseError Enum for storing euccess of parsing.

Enumerator

NoError Parsing was successfull.

OpenFileError Could not open File to parse.

ParseError An Error occured while parsing.

3.3.3 Member Function Documentation

3.3.3.1 int Overlay::frame (quint32 *timestamp*)

Overlay::frame returns the next lower framenumber that comes before timestamp This is done in logarithmic time.

Parameters

timestamp	used as a threshold

Returns

framenumber, always between 0 and the amount of parsed Frames

3.3.3.2 QPair< QVector2D, qint64 > Overlay::nextOverlay (quint32 *timeStamp*)

Overlay::nextOverlay gets the next overlay after timeStamp.

Parameters

timeStamp	

Returns

position and timestamp of the next overlay after timeStamp If there is no next Overlay after timeStamp the function returns -1 as timeStamp

3.3.3.3 QPair< QVector2D, qint64 > Overlay::nextOverlay ( )

Overlay::nextOverlay gets the next overlay after a timeStamp that is internaly stored.

Returns

position and timestamp of the next overlay after timeStamp If there is no next Overlay after an internal stored timestamp the function returns -1 as timeStamp

3.3.3.4 QVector< QVector2D > Overlay::overlaysToFrame (int *frame*)

Overlay::overlaysToFrame.

Parameters

frame	framenumber

Returns

All overlays that have a timestamp between the timestamp of frame and the next frame as a QVector

3.3.3.5 Overlay::ParserError Overlay::parseFrames (QString *fileName*)

Overlay::parseFrames parses the content of fileName as timestamp. Each line that has more or equal to 1 tab separated values and doesn start with '#' is parsed.

Parameters

fileName	Path to the file that is to be parsed

Returns

ParserError OpenFileError when it isn't possible to open the file ParseError when the parser failed to parse any line or a line contained invalid characters Noerror when the parser could parse the file successfuly

3.3.3.6 Overlay::ParserError Overlay::parseOverlays (QString *fileName*)

Overlay::parseOverlays parses the content of fileName as overlays. Each line that has more or equal to 3 tab separated values and doesn start with '#' is parsed.

Parameters

fileName	Path to the file that is to be parsed

Returns

ParserError OpenFileError when it isn't possible to open the file ParseError when the parser failed to parse any line or a line contained invalid characters Noerror when the parser could parse the file successfuly

3.3.3.7 QPair< QVector2D, qint64 > Overlay::previousOverlay (quint32 *timeStamp*)

Overlay::previousOverlay gets the previous overlay before timeStamp.

Parameters

timeStamp	

Returns

position and timestamp of the previous overlay before timeStamp If there is no previous Overlay before time←Stamp the function returns -1 as timeStamp

3.3.3.8 QPair< QVector2D, qint64 > Overlay::previousOverlay ( )

Overlay::previousOverlay gets the previous overlay before a timeStamp that is internaly stored.

Returns

position and timestamp of the previous overlay before timeStamp If there is no previous Overlay before an internal stored timestamp the function returns -1 as timeStamp

3.3.3.9 qint64 Overlay::timeStamp (int *frame*)

Overlay::timeStamp returns the timestamp to frame in constant time.

Parameters

frame	

Returns

timestamp of frame with framenumber frame, -1 if frame is not available

The documentation for this class was generated from the following files:

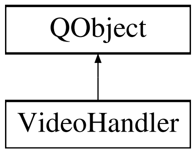
- overlay.h
- overlay.cpp

3.4 VideoHandler Class Reference

The VideoHandler class handles when each frame and overlay gets displayed on screen synchronized, handles the video buffer, requests frames as needed, allows controll of the playback.

```
#include <video_handler.h>
```

Inheritance diagram for VideoHandler:



Public Types

- enum PlayState { pause, playVideo, playOverlays }

  The PlayState enum holds if state of the video playback.

Public Slots

- void play ()

  VideoHandler::play starts video playback with normal speed If the video is already playing it pauses the playback.
- void playOverlay ()

  VideoHandler::playOverlay starts the playback of the overlays with the speed of the normal video. If the overlays are already playing it will pause the playback.
- void timeout ()

  VideoHandler::timeout slot that gets called if a new frame or overlay needs to be displayed. Reduces the buffer size to 50 Images if the buffer is bigger than 100 Images.
- void open ()

  VideoHandler::open opens a new video file and deletes all previous buffered images.
- void openOverlay ()

  VideoHandler::openOverlay opens a csv file and parses it.
- void openTimestamp ()

  VideoHandler::openTimestamp Opens timestamp file and parses the file.
- void imageFreed ()

  VideoHandler::imageFreed slots that gets calles when _currentImage and _previousImage are removed from the scene. And it deletes them.
- void nextImage ()

  VideoHandler::nextImage slot that sends the next image and updates the overlay.
- void nextOverlay ()

  VideoHandler::nextOverlay slot that sends the next Overlay and updates the Image if needed.
- void previousImage ()

  VideoHandler::previousImage slot that sends the previous Image and updates the overlay.
- void previousOverlay ()

  VideoHandler::previousOverlay slot that sends the previous Overlay and updates the Image if needed.

Signals

- void updateSlider (int totalFrames)

  updateSlider emitted when the slider needs to be updated
- void imageRefresh (QGraphicsPixmapItem ∗image)

  imageRefresh emitted when the image needs to be updated
- void overlayRefresh (QPoint pos)

  overlayRefresh emitted when the overlay needs to be adjusted
- void decode (int images)

  decode order new frame(s) to be decoded and converted
- void freeImage ()

  freeImage request ownership of images that are displayed

Public Member Functions

- VideoHandler ()
  VideoHandler::VideoHandler.
- double posMSec () const
  VideoHandler::posMSec.
- double posFrames () const
  VideoHandler::posFrames.
- void setPosFrames (double frame, bool updateCurrentFrame=true)
  VideoHandler::setPosFrames.
- double posRelative () const
  VideoHandler::posRelative.
- double framerate () const
  VideoHandler::framerate.
- double codec () const
  VideoHandler::codec.
- double frameCount () const
  VideoHandler::frameCount.
- PlayState playState () const
  VideoHandler::playState returns the current playState of the video playback.

3.4.1 Detailed Description

The VideoHandler class handles when each frame and overlay gets displayed on screen synchronized, handles the video buffer, requests frames as needed, allows controll of the playback.

3.4.2 Member Enumeration Documentation

3.4.2.1 enum VideoHandler::PlayState

The PlayState enum holds if state of the video playback.

Enumerator

pause Video and overlay is paused.
playVideo Video is playing.
playOverlays Overlay is playing.

3.4.3 Member Function Documentation

3.4.3.1 double VideoHandler::codec ( ) const

VideoHandler::codec.

Returns

Codec of the video that is currently decoded

3.4.3.2 void VideoHandler::decode (int *images*) [signal]

decode order new frame(s) to be decoded and converted

Parameters

images	count of frames to be decoded

3.4.3.3 double VideoHandler::frameCount ( ) const

VideoHandler::frameCount.

Returns

The total amount of frames in the current playback

3.4.3.4 double VideoHandler::framerate ( ) const

VideoHandler::framerate.

Returns

Framerate of the video that is currently decoded

3.4.3.5 void VideoHandler::imageRefresh (QGraphicsPixmapItem * *image*) [signal]

imageRefresh emitted when the image needs to be updated

Parameters

image	

3.4.3.6 void VideoHandler::overlayRefresh (QPoint *pos*) [signal]

overlayRefresh emitted when the overlay needs to be adjusted

Parameters

pos	

3.4.3.7 VideoHandler::PlayState VideoHandler::playState ( ) const

VideoHandler::playState returns the current playState of the video playback.

Returns

3.4.3.8 double VideoHandler::posFrames ( ) const

VideoHandler::posFrames.

Returns

Current positon of the playback in frameposition

3.4.3.9 double VideoHandler::posMSec ( ) const

VideoHandler::posMSec.

Returns

Current position of the playback in milliseconds

3.4.3.10 double VideoHandler::posRelative ( ) const

VideoHandler::posRelative.

Returns

Current positon of the playback relative to the entire playback

3.4.3.11 void VideoHandler::setPosFrames (double *frame*, bool *updateCurrentFrame* = `true`)

VideoHandler::setPosFrames.

Parameters

frame	Frame to set the DecodeWorker to.
updateCurrentFrame	Set

3.4.3.12 void VideoHandler::updateSlider (int *totalFrames*) `[signal]`

updateSlider emitted when the slider needs to be updated

Parameters

totalFrames	total count of frames

The documentation for this class was generated from the following files:

- video_handler.h
- video_handler.cpp

Index