GRAPHREC

PROGRAMMER'S DOCUMENTATION

Version 1.0.0

Prepared by Petr Koupý <petr.koupy@gmail.com>

Last Updated on April 30, 2010

Table of Contents

1 Compilation

GraphRec is dependent on Qt framework and FFmpeg video library, both of which are large enough that it is not reasonable to include their source code into the redistributable package. Once those prerequisites are prepared and configured, GraphRec can be built solely from files that are included in the redistributable package. Since both Qt and FFmpeg are multiplatform projects, it should be possible to perform the compilation on all major platforms. However, testing was done only on Windows and Linux (specifically Windows XP, Windows Vista, Windows 7, Ubuntu 8.10, Ubuntu 9.04, Ubuntu 9.10). Steps in the following guide produce the statically linked build of GraphRec compiled by GNU compilers. Compilation steps are configured to be compatible with processors containing MMX and SSE extensions to the instruction set, which are present on the majority of modern processors. More uncommon extensions (e.g. 3DNow!, SSE2) are disabled. Note that all proposed paths can be changed, however they must not contain any spaces.

1.1 Preparing Environment

Windows

Download following packages from http://sourceforge.net/projects/mingw/files/:

```
binutils-2.19.1-mingw32-bin.tar.gz
mingwrt-3.15.2-mingw32-dll.tar.gz
mingwrt-3.15.2-mingw32-dev.tar.gz
mingw32-make-3.81-20080326-3.tar.gz
gcc-core-4.4.0-mingw32-bin.tar.gz
gcc-core-4.4.0-mingw32-dll.tar.gz
gcc-c++-4.4.0-mingw32-bin.tar.gz
gcc-c++-4.4.0-mingw32-dll.tar.gz
w32api-3.13-mingw32-dev.tar.gz
pthreads-w32-2.8.0-mingw32-dll.tar.gz
gmp-4.2.4-mingw32-dll.tar.gz
libiconv-1.13-mingw32-dll-2.tar.gz
mpfr-2.4.1-mingw32-dll.tar.gz
```

Unpack all downloaded packages into `c:\mingw\` and confirm all overwrite warnings.

Linux

Make sure you have installed following packages and their dependencies from repositories:

```
binutils
binutils-static
make
gcc
libgcc
libc6
libc6-dev
libglib
libglib-dev
cpp
g++
libstdc++6
libstdc++6-dev
libfontconfig
libfontconfig-dev
libxrender
libxrender-dev
```

Note: Although the goal is to create statically linked executable, on Linux it is dangerous to statically link against system libraries. Thus, the resulting executable will be linked statically only against Qt and FFmpeg. Since libraries like `libc6` or `libstdc++6`, both of which will be linked dynamically, are not backwards compatible, it is advised to carry out the build process with older versions of listed packages. On the other hand, mentioned libraries are forwards

compatible up until now. Therefore, the older the packages will be, the more systems will be able to run the executable. However, there must be done a trade-off between the amount of compatible systems and the efficiency of produced code, assuming the newer versions of libraries remove bugs and improve performance.

1.2 Building Qt

Windows

Download `qt-everywhere-opensource-src-4.6.2.zip` from http://get.qt.nokia.com/qt/source/.

Unpack the archive into `c:\qt\`.

Edit the file `c:\qt\mkspecs\win32-g++\qmake.conf` by inserting `-static -static-libgcc` to the beginning of `QMAKE_LFLAGS`. Resulting line should be:

```
QMAKE_LFLAGS = -static -static-libgcc -enable-stdcall-fixup -Wl,-enable-auto-
import -Wl,-enable-runtime-pseudo-reloc
```

Execute the following batch script from within `c:\qt\`:

```
PATH = %PATH%;c:\mingw\bin\;c:\qt\bin\
configure.exe -nomake tools -nomake examples -nomake demos -nomake docs -
nomake translations -release -opensource -confirm-license -static -ltcg -fast
-no-exceptions -no-accessibility -stl -no-sql-mysql -no-sql-psql -no-sql-oci -
no-sql-odbc -no-sql-tds -no-sql-db2 -no-sql-sqlite -no-sql-sqlite2 -no-sql-
ibase -no-qt3support -no-opengl -no-openvg -qt-zlib -no-gif -qt-libpng -no-
libmng -no-libtiff -qt-libjpeg -no-dsp -no-vcproj -no-incredibuild-xge -
plugin-manifests -qmake -process -rtti -mmx -no-3dnow -sse -no-sse2 -no-
openssl -no-dbus -no-phonon -no-multimedia -no-audio-backend -no-webkit -no-
script -no-scripttools -no-declarative -no-style-plastique -no-style-
cleanlooks -no-style-motif -no-style-cde -no-native-gestures -no-iwmmxt -no-
crt -no-cetest -no-freetype -no-s60
qmake.exe projects.pro -o Makefile -spec win32-g++
mingw32-make.exe
```

Linux

Download `qt-everywhere-opensource-src-4.6.2.tar.gz` from http://get.qt.nokia.com/qt/source/.

Unpack the archive into `/tmp/qt/`.

Execute the following shell script from within `/tmp/qt/` with elevated privileges:

```
#!/bin/sh
./configure -nomake tools -nomake examples -nomake demos -nomake docs -nomake
translations -release -opensource -confirm-license -static -fast -no-
exceptions -no-accessibility -stl -no-sql-mysql -no-sql-psql -no-sql-oci -no-
sql-odbc -no-sql-tds -no-sql-db2 -no-sql-sqlite -no-sql-sqlite2 -no-sql-
sqlite_symbian -no-sql-ibase -no-qt3support -no-xmlpatterns -no-multimedia -
no-audio-backend -no-phonon -no-phonon-backend -no-webkit -no-javascript-jit -
no-script -no-scripttools -no-declarative -no-3dnow -no-sse2 -qt-zlib -no-gif
-no-libtiff -qt-libpng -no-libmng -qt-libjpeg -no-openssl -no-nis -no-cups -
no-iconv -no-dbus -no-gtkstyle -no-nas-sound -no-opengl -no-openvg -no-sm -
xshape -xsync -no-xinerama -no-xcursor -no-xfixes -no-xrandr -xrender -mitshm
-fontconfig -no-xinput -no-xkb –glib
make
make install
```

1.3 Building FFmpeg

Windows

Create following directories:

```
c:\msys\
c:\msys\bin\
c:\msys\mingw\
```

Download following packages from http://sourceforge.net/projects/mingw/files/:

```
binutils-2.19.1-mingw32-bin.tar.gz
mingwrt-3.15.2-mingw32-dll.tar.gz
mingwrt-3.15.2-mingw32-dev.tar.gz
gcc-core-3.4.5-20060117-1.tar.gz
gcc-g++-3.4.5-20060117-1.tar.gz
w32api-3.13-mingw32-dev.tar.gz
```

Unpack all downloaded packages into `c:\msys\mingw\` and confirm all overwrite warnings.

Download `coreutils-5.97-2-msys-1.0.11-bin.tar.lzma` and `coreutils-5.97-2-msys-1.0.11-ext.tar.lzma` from http://sourceforge.net/projects/mingw/files/.

Unpack the archives into `c:\msys\bin\`.

Download `MSYS-1.0.11.exe` from http://sourceforge.net/projects/mingw/files/.

Run the installer and set the installation path to `c:\msys\`. Installation will be automatically finished the by post-installation batch script, where upon request the MinGW path must be set to `c:/msys/mingw` (note forward slashes).

Download `ffmpeg-0.5.tar.bz2` from http://ffmpeg.org/releases/.

Unpack the archive into `c:\ffmpeg\`.

Run the *MSYS* environment from the Start menu, change the directory to `c:/ffmpeg/` (note forward slashes) and execute the following bash script:

```
#!/bin/sh
./configure --enable-gpl --disable-ffmpeg --disable-ffplay --disable-ffserver
--enable-swscale --disable-vhook --disable-network --disable-ipv6 --disable-
mpegaudio-hp --enable-memalign-hack --disable-encoders --enable-encoder=flv --
enable-encoder=h263 --enable-encoder=h263p --enable-encoder=mpeg1video --
enable-encoder=mpeg2video --enable-encoder=mpeg4 --enable-encoder=rv10 --
disable-decoders --disable-muxers --enable-muxer=avi --enable-muxer=flv --
enable-muxer=h263 --enable-muxer=matroska --enable-muxer=mov --enable-
muxer=mp4 --enable-muxer=mpeg1system --enable-muxer=mpeg1vcd --enable-
muxer=mpeg1video --enable-muxer=mpeg2dvd --enable-muxer=mpeg2svcd --enable-
muxer=mpeg2video --enable-muxer=rm --enable-muxer=swf --enable-muxer=tgp --
disable-demuxers --disable-parsers --disable-bsfs --disable-protocol=pipe --
disable-devices --disable-filters --disable-altivec --disable-amd3dnow --
disable-amd3dnowext --disable-mmx2 --disable-ssse3 --disable-armv5te --
disable-armv6 --disable-armv6t2 --disable-armvfp --disable-iwmmxt --disable-
mmi --disable-neon --disable-vis --disable-debug
make
make install
```

Copy `c:\msys\local\include` to `c:\qt\include`.

Copy `c:\msys\local\lib` to `c:\qt\lib`.

Linux

Download `ffmpeg-0.5.tar.bz2` from http://ffmpeg.org/releases/.

Unpack the archive into `/tmp/ffmpeg/`.

Execute the following shell script from within `/tmp/ffmpeg/` with elevated privileges:

```
#!/bin/sh
./configure --enable-gpl --disable-ffmpeg --disable-ffplay --disable-ffserver
--enable-swscale --disable-vhook --disable-network --disable-ipv6 --disable-
mpegaudio-hp --enable-memalign-hack --disable-encoders --enable-encoder=flv --
enable-encoder=h263 --enable-encoder=h263p --enable-encoder=mpeg1video --
enable-encoder=mpeg2video --enable-encoder=mpeg4 --enable-encoder=rv10 --
disable-decoders --disable-muxers --enable-muxer=avi --enable-muxer=flv --
enable-muxer=h263 --enable-muxer=matroska --enable-muxer=mov --enable-
muxer=mp4 --enable-muxer=mpeg1system --enable-muxer=mpeg1vcd --enable-
muxer=mpeg1video --enable-muxer=mpeg2dvd --enable-muxer=mpeg2svcd --enable-
muxer=mpeg2video --enable-muxer=rm --enable-muxer=swf --enable-muxer=tgp --
disable-demuxers --disable-parsers --disable-bsfs --disable-protocol=pipe --
disable-devices --disable-filters --disable-altivec --disable-amd3dnow --
disable-amd3dnowext --disable-mmx2 --disable-ssse3 --disable-armv5te --
disable-armv6 --disable-armv6t2 --disable-armvfp --disable-iwmmxt --disable-
mmi --disable-neon --disable-vis --disable-debug
make
make install
```

1.4 Building GraphRec

Windows

Download `GraphRec-1.0.0-Win32.zip` from http://koupy.net/download/.

Unpack the archive into `c:\graphrec\`.

Execute the following batch script from within `c:\graphrec\src\`:

```
PATH = %PATH%;c:\mingw\bin\;c:\qt\bin\
qmake.exe GraphRec.pro -spec win32-g++ -r CONFIG+=release CONFIG+=static
QTPLUGIN+=qjpeg DEFINES+=G_GRSTATIC
mingw32-make.exe
```

Resulting executable is `c:\graphrec\src\release\GraphRec.exe`.

Linux

Download `GraphRec-1.0.0-X11.tgz` from http://koupy.net/download/.

Unpack the archive into `/temp/graphrec/`.

Execute the following shell script from within `/temp/graphrec/src/`:

```
#!/bin/sh
PATH=/usr/local/Trolltech/Qt-4.6.2/bin:$PATH
export PATH
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH
qmake GraphRec.pro -spec linux-g++ -r CONFIG+=release CONFIG+=static
```

```
QTPLUGIN+=qjpeg DEFINES+=G_GRSTATIC
make
```

Resulting executable is `/tmp/graphrec/src/GraphRec`.

1.5 Compressing Executable

Windows

Download `upx304w.zip` from http://upx.sourceforge.net/download/.

Unpack the archive into `c:\upx\`.

Copy `c:\graphrec\src\release\GraphRec.exe` to `c:\upx\`.

Execute the following batch script from within `c:\upx\`:

```
upx.exe --best --lzma GraphRec.exe
```

Resulting executable is `c:\upx\GraphRec.exe`.

Linux

Download `upx-3.04-i386_linux.tar.bz2` from http://upx.sourceforge.net/download/.

Unpack the archive into `/tmp/upx/`.

Copy `/tmp/graphrec/src/GraphRec` to `/tmp/upx/`.

Execute the following shell script from within `/tmp/upx/`:

```
upx --best --lzma GraphRec
```

Resulting executable is `/tmp/upx/GraphRec`.

1.6 Redistributable Package

Package structure:

- `bin` folder contains the main executable.
- `src` folder contains sources additional files (images, icons) that are needed for compilation.
- `doc` folder contains documentation files. GraphRec is looking for the entry point named *index.html*.
- `samples` folder contains input data for testing purposes.

Both packages, for Windows and Linux, contain the application launcher in their top-level directory. Launcher is simple batch/shell script that runs the main executable located in `bin` directory.

The installer is made by the third-party software called *Install Jammer* (http://www.installjammer.com/). It is free, cross-platform install builder with high level of configurability. It supports both self-unpacking installers and archives.

1.7 Licensing

Both Qt and FFmpeg are licensed under the dual license – either GNU GPL or GNU LGPL. GraphRec uses FFmpeg *swscale* support for the highly optimized conversion between RGB and YUV images. Since *swscale* support is GPL-only, GraphRec must be also licensed under GPL. This implies that redistributable package of GraphRec must contain complete source code and full text of GNU GPL

license. GPL license also demands that each source code file begins with the license stub. By the viral nature of GPL license, all changes and additions to GraphRec must be released under the same or less restrictive license. Concerning the packed executable, UPX is also licensed under GPL. However, as stated on the project website, UPX decompression stub inserted into the compressed executable is not a subject to the GPL.

2 Algorithms

Purpose of this section is to explain some techniques and algorithms used in GraphRec. Description is rather abstract in order to support notion over actual implementation.

2.1 Multirobot Validation

Reference: *multirobotvalidator.cpp*, *pebblevalidator.cpp*

GraphRec reads arbitrary number of entity moves from the input file. Each move is specified only by its source node, destination node and time step at which occurs. There is no assumption about how these moves interact with each other. Since some moves can be in conflict, they have to be validated against definition of either *Multirobot path planning* or *Pebble motion on graph*:

1) All moves are sorted by the time step with quicksort algorithm.
2) Moves are split into groups that are characterized by the same time step. Each group is then filtered. Moves that are **discarded** meet one of these criteria:
    a) Source node is equal to destination node, which effectively results in a loop.
    b) There is no edge between source and destination node.
    c) Source node does not contain entity in the respective time step.
    d) There is already different approved move that begins in the same source node as assessed move.
    e) There is already different approved move that ends in the same destination node as assessed move.
    f) There is already different approved node that begins in the destination node and ends in the source node of assessed move (inverse move).
3) Second pass through each of filtered groups determines final valid moves. Moves that are **approved** meet one of these criteria:
    a) Source node contains entity and destination node is empty.
    b) (**only** *Multirobot path planning*) Both source node and destination node contain entity. Recursive search proves that destination node is freed by some other move occurring in the same time step and that whole chain of such moves is terminated by an empty node.

Note: For those, who find themselves studying source code or input file format in detail, it should be stated that arrow displayed in each movement definition is not always indicating actual direction of the movement. Because of this ambiguity, it is not clear, which node should be considered as a source. This inconveniency is caused by file format that is shared between GraphRec and planning software, which validation was written for. In order to resolve this issue, direction is determined by the presence of entity. If entity is present in a node pointed by the arrow and the other node is empty, and if movement is otherwise valid, it can occur even in the opposite direction than the one suggested by arrow. This is perfectly unambiguous in the case of isolated moves (because inverse movement would be otherwise considered as invalid). However, when it comes to chain moves, which are allowed by *Multirobot path planning*, ambiguity is causing problems again. In the situation where two mutually inverse chains with common first movement were recognized as valid, it would be impossible to decide which one should be approved. Thus, current implementation analyses only chains that correspond with arrow direction.

2.2 Fruchterman-Reingold Layouting

Reference: *fruchtermanreingoldlayouter.cpp*

Fruchterman-Reingold layouting algorithm is one of so-called *force-directed* layouting algorithms. It is based on the idea that nodes are **repulsive particles** and edges are **contracting springs**. Target distance between nodes is proportional to their **adjacency**. Algorithm is iterative gradually providing better and better layout for **all nodes at each cycle**. Utilized force model is **elastic**, so

that user can directly interact with running algorithm. GraphRec uses custom modified version of this algorithm. Pseudocode:

in $G(V,E)$ {initial positions of nodes are random}
in $disp$ {target displacement of nodes}
forever
  {calculate new positions}
  foreach $v = (x, y, acc_x, acc_y) \in V$ **do**
    $v.acc_x \leftarrow 0$ {accumulator for horizontal position change}
    $v.acc_y \leftarrow 0$ {accumulator for vertical position change}
    {accumulate repulsive forces}
    foreach $u = (x, y, acc_x, acc_y) \in V \setminus \{v\}$ **do**
      let $d_x \leftarrow |v.x - u.x|$
      let $d_y \leftarrow |v.y - u.y|$
      let $d \leftarrow d_x^2 + d_y^2$
      let $f_r \leftarrow disp^2/\sqrt{d}$ {repulsive force multiplier}
      $v.acc_x \leftarrow v.acc_x + (d_x \cdot f_r)/d$
      $v.acc_y \leftarrow v.acc_y + (d_y \cdot f_r)/d$
    {accumulate attractive forces}
    foreach $e = (m,n) \in E_v = \{e \in E \mid \exists w \in V : e = (v,w) \vee e = (w,v)\}$ **do**
      let $d_x \leftarrow |e.m.x - e.n.x|$
      let $d_y \leftarrow |e.m.y - e.n.y|$
      let $d \leftarrow d_x^2 + d_y^2$
      let $f_a \leftarrow d/disp$ {attractive force multiplier}
      $v.acc_x \leftarrow v.acc_x - (|E_v| \cdot d_x \cdot f_a)/d$
      $v.acc_y \leftarrow v.acc_y - (|E_v| \cdot d_y \cdot f_a)/d$
  {update positions}
  let $static \leftarrow 0$ {counter for nodes with insignificant change}
  foreach $v = (x, y, acc_x, acc_y) \in V$ **do**
    if $v.acc_x < \varepsilon \wedge v.acc_y < \varepsilon$ **then**
      $static \leftarrow static + 1$
    else
      $v.x \leftarrow v.x + v.acc_x$
      $v.y \leftarrow v.y + v.acc_y$
  if $static = |V|$ **then**
    break

Time complexity of single iteration is $O(|V|^2 + |E|)$.Original algorithm is terminated by *simulated annealing* method, which gradually weakens the forces and algorithm is thus focusing more and more on cosmetic rather than radical changes. Unfortunately, simulated annealing is not suitable for interactive layouting, because it is slower and less intuitive. Therefore, modified algorithm is terminated simply by checking when all vertices are moved only insignificantly.

Other deviations from the original algorithm are rather minor. Modified algorithm emphasizes dependency between attractive force and number of edges that are connected to the vertex. Since attractive force is multiplied by the number of connected edges, poorly connected vertices on the outer boundary of the graph are not so much attracted by adjacent nodes. This approach gives better layout especially in the case of regular grid, where boundary nodes tend to be dragged to the center of the graph. Another difference is that modified algorithm does not calculate some squaring and root extraction – algorithm then behaves more dynamically while user is dragging some node (graph follows dragged node better). It should be noted that these modifications were made in the trial and error style.

2.3 Kamada-Kawai Layouting

Reference: *kamadakawailayouter.cpp*

Kamada-Kawai is also iterative force-directed layouting algorithm. Target distance between nodes is proportional to the graph-theoretic distance (**shortest path**). Single iteration improves position of only **one node at a time**. When calculating its position, all other nodes are considered as solid anchors for springs hooked together in the current location of processed node. **Springs are both**

contractive/repulsive depending on their current stretch and their equilibrium, which is proportional to the shortest path between the two nodes (thus, springs have different strengths). Algorithm is **non-elastic** and thus not very interactive. Kamada-Kawai algorithm utilizes several methods and principles from very different fields (differential calculus, linear algebra, graph theory, material mechanics).

All-pairs shortest-paths problem can be solved in $O(|V|^3)$ by *Floyd-Warshall* algorithm:

in $G(V,E)$
let $D = \{d_{ij}\} \leftarrow \begin{cases} 0, & i = j \\ 1, & v_i, v_j \in V, (v_i, v_j) \in E \\ \infty, & v_i, v_j \in V, (v_i, v_j) \notin E \end{cases}$
for $k \leftarrow 1, \dots, |V|$ **do**
  for $i \leftarrow 1, \dots, |V|$ **do**
    for $j \leftarrow 1, \dots, |V|$ **do**
      $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$

Spring counteracts its elongation/contraction by the force $F(x) = K(x - x_0)$, which is linearly proportional to its deflection x from the equilibrium x_0. Accumulated potential energy corresponds to the integral of force:

$$E(x) = \int F(x)\,dx = \frac{1}{2}K(x - x_0)^2$$

Factor K is defined for nodes $v_i, v_j \in V$ as $k_{ij} = K_0/d_{ij}^2$, where K_0 is arbitrary constant. Thus, spring is more solid between closer nodes (because of shorter path). Similarly, equilibrium length of the spring is defined as $l_{ij} = disp \cdot d_{ij}$, where $disp$ is external parameter specifying target displacement of nodes. Finally, considering set of nodes $|V| = n$, where $v_i \in V$ has Euclidian coordinates x_i, y_i, the potential energy of the whole system can be defined as sum of potential energies for all springs:

$$E(x_1, \dots, x_n, y_1, \dots, y_n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - l_{ij} \right)^2$$

At each iteration, algorithm chooses node v_m with the maximum size of the gradient vector ∇E_m, where $E_m(x_m, y_m)$ is function E whose variables (apart from x_m, y_m) are considered as fixed constants:

$$\nabla E_m = \left(\frac{\partial E}{\partial x_m}, \frac{\partial E}{\partial y_m} \right), |\nabla E_m| = \sqrt{\left(\frac{\partial E}{\partial x_m} \right)^2 + \left(\frac{\partial E}{\partial y_m} \right)^2}$$

$$\frac{\partial E}{\partial x_m} = \sum_{i=1}^{n} k_{mi} \left((x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\sqrt{(x_m - x_i)^2 + (y_m - y_i)^2}} \right)$$

$$\frac{\partial E}{\partial y_m} = \sum_{i=1}^{n} k_{mi} \left((y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\sqrt{(x_m - x_i)^2 + (y_m - y_i)^2}} \right)$$

Gradient vector is then gradually decreased by moving the node to the local minimum of E_m. *Newton-Raphson* numerical method approximates zero of the function $f(x)$ by iterating through the equation $x_{n+1} = x_n - f(x_n)/f'(x_n)$, which can be also expressed as $f'(x_n) \cdot dx = -f(x_n)$, where $dx = (x_{n+1} - x_n)$. Let us apply this method to approximate the zero of gradient vector ∇E_m, effectively finding the local minimum of E_m. This includes calculating the *Jacobian* of ∇E_m:

$$J_{\nabla E_m} \cdot \begin{pmatrix} dx \\ dy \end{pmatrix} = -\nabla E_m$$

$$\begin{bmatrix} \frac{\partial^2 E}{\partial x_m^2} & \frac{\partial^2 E}{\partial x_m \partial y_m} \\ \frac{\partial^2 E}{\partial y_m \partial x_m} & \frac{\partial^2 E}{\partial y_m^2} \end{bmatrix} \cdot \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} -\frac{\partial E}{\partial x_m} \\ -\frac{\partial E}{\partial y_m} \end{bmatrix}$$

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i=1}^{n} k_{mi} \left(1 - \frac{l_{mi}(y_m - y_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}\right)$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i=1}^{n} k_{mi} \left(\frac{l_{mi}(x_m - x_i)(y_m - y_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}\right)$$

$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i=1}^{n} k_{mi} \left(1 - \frac{l_{mi}(x_m - x_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}\right)$$

In order to solve above system of two linear equations, let us use *Cramer's rule*:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix}$$

$$x = \frac{ed - bf}{ad - bc}, y = \frac{af - ec}{ad - bc}$$

Finally, the pseudocode for Kamada-Kawai can be written in the following way:

in $G(V, E)$
in $D = \{d_{ij}\}$ {shortest paths}
in $L = \{l_{ij}\}$ {equilibrium lengths}
in $K = \{k_{ij}\}$ {strength of springs}
while $max_{i=1}^{n}(|\nabla E_i|) > \varepsilon$ **do**
    let $\nabla E_m \leftarrow \nabla E_j : |\nabla E_j| = \max_{i=1}^{n}(|\nabla E_i|), j = 1, \dots, n$
    let $v \leftarrow v_i \in V : i = m, i = 1, \dots, n$
    while $|\nabla E_m| > \varepsilon$ **do**
        {solve the system $J_{\nabla E_m} \cdot \begin{pmatrix} dx \\ dy \end{pmatrix} = -\nabla E_m$ }
        $v.x \leftarrow v.x + dx$
        $v.y \leftarrow v.y + dy$

Function E is at its local minimum, when all of its first partial derivatives are equal to zero. Algorithm approximates this target by gradually decreasing greatest ∇E_i gradient vectors, one at a time, until all vectors have its elements (first partial derivatives) sufficiently close to zero. Partial derivative can be computed in $O(|V|)$. Outer loop calculates $|V|$ gradients, each consisting of 2 partial derivatives, in $O(|V|^2)$. Inner loop calculates 4 derivatives for Jacobian and 2 derivatives in order to update ∇E_m. Since count T of inner loop iterations (Newton-Raphson method) depends untrivially on node count, node positions and graph structure, resulting cost of inner loop for one outer loop iteration is $O(T|V|)$. It should be noted that, when several conditions hold, Newton-Raphson method is proved to converge quadratically to the zero of the given function. However, convergence might fail when those condition are not met (e.g. the initial value is too far from zero). Thus, GraphRec puts the upper limit on the number of inner loop executions to prevent lock-ups, which effectively means that the overall cost of inner loop for one outer loop iteration is only $O(|V|)$. Hence the complexity of single iteration of the outer loop is $O(|V|^2)$. Note that the very first iteration is $O(|V|^3)$ due to the Floyd-Warshall.

2.4 Producer-consumer Synchronization

Reference: *main.cpp*, *graphview.cpp*, *ffmpegvideorecorder.cpp*

GraphRec uses concurrency while encoding video. Rendering and encoding is done on different threads. Thus, rendering function acts as a producer, who puts video frames into the buffer of limited size, and encoding function acts as a consumer, who reads frames from the buffer and encodes them into output file. Both threads are synchronized by two semaphores:

in $buffer$ {limited size}
in $free$ {semaphore guarding empty part of buffer}

in $used$ {semaphore guarding occupied part of buffer}
let $free.count \leftarrow buffer.size$
let $used.count \leftarrow 0$

procedure Producer
    acquire ($free$)
    {put data into buffer}
    release ($used$)

procedure Consumer
    acquire ($used$)
    {read data from buffer}
    release ($free$)

Function for acquiring semaphore normally blocks until resources are available. Since GraphRec must stay responsive during video encoding, acquiring semaphore is periodically requested only for small amount of time after which application message queue is inspected:

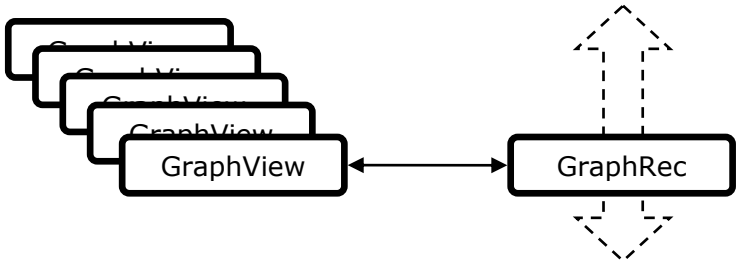
while tryacquire ($semaphore, timeout$) $= false$ **do**
    {process application messages/events}
{read data from buffer}
release ($free$)

3 Architecture

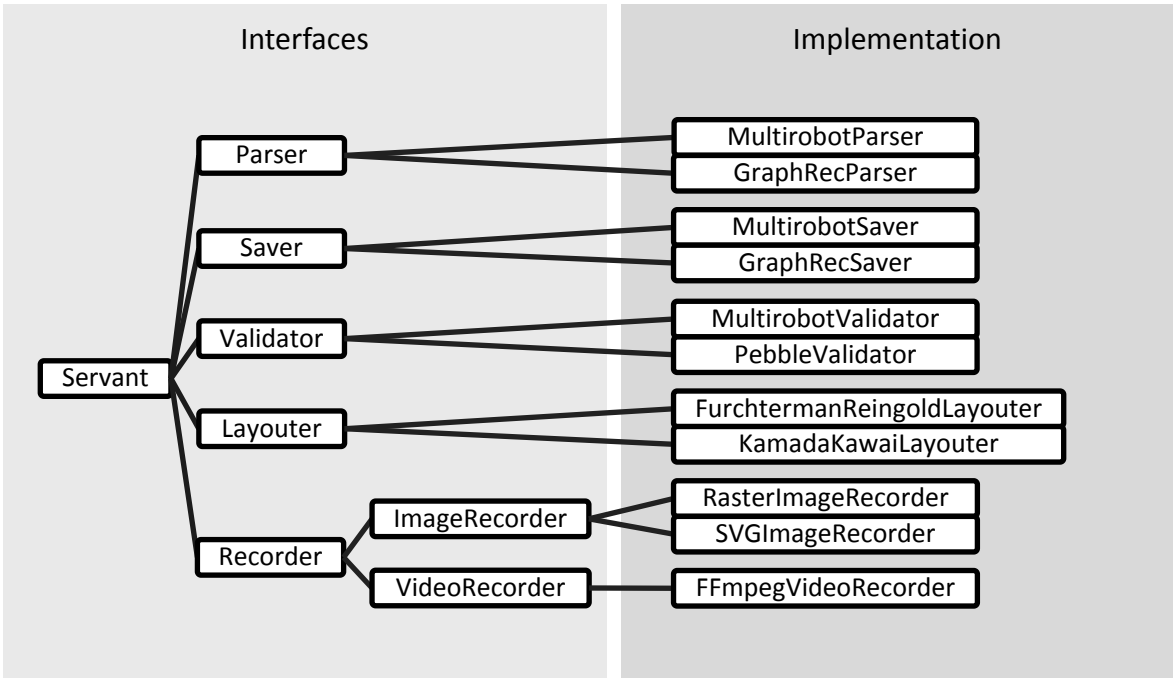
Knowledge of architecture is critical for any modification or expansion of GraphRec source code. First subsection briefly describes whole GraphRec in a top-down fashion. Overview should be read before any other subsection, because these are ordered inversely (bottom-up) and thus not providing any hindsight. In each subsection, it is assumed that reader knows information from previous subsections. For those who are interested in making additional modules for validating, layouting or file handling, it is sufficient to read only first three subsections. It is expected that reader will be reviewing reference source code while going through subsections. Before reading any architecture section, it is also recommended to read corresponding section in the *User's Guide*. Since it is not in the scope of this manual, it is assumed that reader understands specific aspects of Qt programming – mainly signal/slot mechanism and GUI designing. It should be only noted here, that signal/slot mechanism provides a way to construct more flexible architecture at the cost of some performance (sending signal is approximately 10 times slower than normal function call). Qt GUI designing is built around *qmake*, which takes XML document describing GUI and compiles it into C++ code. XML description of GUI can be done manually or with help of QtDesigner. Generated C++ class for GUI is then accessible by a special pointer.

3.1 Global Overview

On the top of the hierarchy, there is `GraphRec` class, which in fact represents the functionality and behavior of the main window and its menu, tool bar and status bar. `GraphRec` also handles file opening/saving and manages a collection of `GraphView` instances. Each `GraphView` stands for one tab that contains a graph. Because `GraphRec` provide user controls, which are shared by all `GraphView` instances, there must be a mechanism for sending user input to the correct `GraphView` (the one which is foreground and focused). This is flexibly achieved by signal/slot mechanism, which allows literally connecting/disconnecting respective `GraphView` to/from `GraphRec`. `GraphView` class covers the rest of the application functionality – validation, layouting, animation and capturing. `GraphView` also stores graph representation and is the owner of almost all dialogs (apart from file handling dialogs that are owned by `GraphRec`). This implies that each `GraphView` has its own set of dialogs and thus its own settings. However, configuration of single `GraphView` can be saved to persistent storage (registry/file) and then it serves as a template for all new instances.



Since `GraphView` responsibility is very wide, it is distributed onto several other classes and dialogs. In order to pass data among these classes, there is common data structure `Context`. It serves as a container for several collections containing graph representation, movement calendar and some settings. Idea is that all those classes alter one common `Context`, which is then displayed by `GraphView`. Data collections in `Context` are assembled from classes that represent primitives – `Node`, `Edge` and `Entity`. Graph itself is represented either by list of `Node` instances, each of which contains list of connected `Edge` instances, or by list of `Edge` instances, each of which contains its `Node` pair.



As was said earlier, `GraphView` utilizes other classes to achieve some tasks. These classes can be split into two groups – dialogs and servants. *Dialog* usually exposes part of the `Context` to the user (e.g. color editing) or provides interface for additional functionality (e.g. video capturing). All dialogs are derived from `QDialog` class, thereby forming simple two-level hierarchy. Servants are more complicated. Each *servant* provides specialized functions that can be called by its owner. Servants form three-level hierarchy – first level is constituted of `Servant` interface, which is further inherited by more specialized second-level interfaces (e.g. `Layouter` interface for graph layouting). Third level is composed from actual implementations of second-level interfaces (e.g. `FruchtermanReingoldLayouter` using specific algorithm for graph layouting). `GraphRec`, `GraphView` and dialogs are programmed against servant interfaces located in the first and second level. Implementations on the third level are identified by their name and evidenced in the `Factory` class. When the owner wants to construct servant of some type (or even a name), its request is passed to the `Factory`, which returns `Servant` pointer to the specified servant. Owner can specify either type or servant name. Described mechanism brings modularity and extensibility into GraphRec source code. Programmer who wants to implement additional servant (e.g. for layouting) needs to know only primitives, `Context` and servant interfaces. Rest of the application is isolated from him.

To provide better understanding of the data flow through application, example of basic user story is described in the following paragraph. Let us say that user want to open, edit and save single solution of *Multirobot path planning*. He/she invokes `OpenDialog` (owned by `GraphRec`) and loads file into it. `OpenDialog` asks its `MultirobotParser` servant to find solutions in the file. After the solution is selected and dialog confirmed, solution location is handed to `GraphView`, which in turn creates its own `MultirobotParser` servant and asks it to initialize `Context` by collecting information from provided file location. Immediately afterwards, `Context` is validated by `MultirobotValidator` servant and finally displayed by `GraphView`. Initialized `GraphView` is connected to `GraphRec` user interface and user can start to do some work. As an example, user might decide to use automatic layouting. In that case, `GraphView` asks `FruchtermanReingoldLayouter` servant to calculate node positions in the `Context`. User also might not be satisfied with graph coloring. Therefore, he/she invokes `ColorDialog`, in which color of any primitive in the `Context` can be changed. On the other hand, animation and rendering is handled by `GraphView` itself. If user decides to save the solution, `GraphRec` creates `MultirobotSaver` servant and provides it to respective `GraphView`, which in turn asks it to save the `Context`.

3.2 Graph Primitives

Graph is represented by three primitives – `Node`, `Edge` and `Entity` – all of which are described in following subsections. Whereas `Entity` is essentially only a data container, both `Node` and `Edge` inherits `QGraphicsItem` and implements its paint event handler. Because paint handlers are called quite frequently and `QGraphicsItem` might need some unrelated data to paint itself (e.g. `Node` needs current time step to infer color), it is reasonable to maintain local copies of those data rather than asking for them at each paint event. Propagation of such information is done via signal/slot mechanism – thus, to continue example, when time step is changed in the `GraphView`, it is also changed in all `Node` instances.

3.2.1 Entity Class

Reference: *entity.h*, *entity.cpp*

`Entity` is a simple class containing its identification (`m_id`), final time step (`m_timestepFinal`) and color information. Final time step is the one after which `Entity` does not move anymore (stops changing owners). Each `Entity` stores its normal (`m_clBackground`, `m_clForeground`) and final (`m_clBackgroundFinal`, `m_clForegroundFinal`) color set. `Node` owning the `Entity` is colored by final color set when and after the final time step is reached. Color set is a pair of colors that are used by hosting `Node` for both background and label. Note that `Entity` itself is actually not visible in the `GraphView` – it only provides information to the hosting `Node`.

3.2.2 Node Class

Reference: *node.h*, *node.cpp*

`Node` is a class inherited from `QGraphicsItem`. It contains its identification (`m_id`), reference to the currently contained `Entity` (`m_entity`) and list of references to connected `Edge` instances (`m_edges`). For purposes of animation, `Node` provides shallow copy constructor `CloneShallow()`. Shallow copy is not connected to the graph and is intended only as a temporary object used for depicting movement of `Entity` between two `Node` instances. Since `Node` is one of the visual elements rendered by `GraphView`, it must be able to paint itself accordingly to its inner state and contained `Entity`:

- In case of null reference to `Entity`, `Node` is colored by its own color set (`m_clBackground`, `m_clForeground`, `m_clBoundary`). Otherwise, it is colored by colors provided by hosted

Entity. Node keeps track of current time step (m_timestep), which is compared at every paint event with final time step discovered from the m_entity.
- Optionally, Node is able to display its identification or even identification of contained Entity. Possible combinations are listed in NodeDescription enumeration and saved in m_description variable.
- Node keeps track of its current position. In the case of discrete positioning (m_discreteDisplacementEnabled), Node snaps itself to the closest allowed location (m_discreteDisplacementOffset). This functionality is implemented in the event handler itemChange() for positional changes and in the AlignPoint() function. Change of position is reported to GraphView via NodePositionChanged() signal.
- At every position change, all Edge instances connected to the Node (m_edges) are requested to update their positions as well (Edge::Adapt()).
- When selected or moved by the user, *Z* coordinate of Node is elevated over the rest of graph elements displayed by GraphView. *Z* coordinate is returned to its original value after the action is finished. Third level of *Z*-axis is reserved for normal Node instances, fourth level for selected instances and finally fifth level is reserved for grabbed instance (the one dragged by mouse). Note that all three levels are defined above the levels reserved for Edge. Also note that there is automatic sub hierarchy between graphic elements that share the same *Z* coordinate. Refer to mousePressEvent() and mouseReleaseEvent().

3.2.3 Edge Class

Reference: *edge.h*, *edge.cpp*

Likewise Node, Edge is another class inherited from QGraphicsItem and rendered by GraphView. It contains references (m_nodeSource, m_nodeDestination) and positions (m_ptSource, m_ptDestination) of its two connected Node instances. Positions are stored for efficiency, since it is assumed that Edge paint event is called more frequently than node positions are changed. Edge must be able to render itself in normal and highlighted mode (m_highlight), which is represented by different color and increased thickness. Each time when the Entity, represented by shallow copy of hosting Node (Node::CloneShallow()), is moving along the trajectory depicted by the respective Edge, shallow copy of this Edge (Edge::CloneShallow()) is switched to highlighted mode (m_highlight) and placed exactly under its prototype on the *Z*-axis. This approach ensures that highlighting thick line will be always painted on the background not overlapping lines of non-highlighted Edge instances. As was stated in Node description, shallow copy is intended only for mentioned purpose. First level of *Z*-axis is reserved for highlighted Edge instances and second level is reserved for normal Edge instances. Note that both levels are defined below the levels reserved for Node.

3.3 Passing Common Data

Reference: *context.h*, *context.cpp*

Context is a class containing common data structures used by GraphView, dialogs and servants. Each Context is owned by GraphView and passed by reference to other objects. Note that Context can be extended by additional data entries. However, current data entries must remain the same, because many classes depend on them. Description of data entries follows:

- Graph is identified by its filePosition in the file of name fileName located in filePath. For convenience, fileName and filePath are joined in the fileCompleteName. Graph can also have a title graphName, which is currently used as a placeholder for graph ID optionally provided by input file. Both graphName and fileName are exposed to the user – graphName as a tab label and fileName as a tab context help.
- Hashed map nodes maps node identifications to respective Node instances.
- Hashed map entities maps entity identifications to respective Entity instances
- Hashed map edges maps pairs of node identifications to respective Edge instances. Intended for fast lookup of Edge instance when only identifications of its two nodes are known.
- CalendarEvent is a structure consisting of timestep and move, which is expressed as a pair of source node identification and destination node identification. CalendarEvent also

contains two flags – `valid` determines whether event was approved by validator and `reverse` determines movement direction. Thus, if the direction suggested by input file is evaluated as invalid, validator has an option to approve inverse direction instead (if it is valid). All `CalendarEvent` structures are stored in an ordered list `calendar`. List is ordered by `timestep` values. Since more than one `CalendarEvent` can have the same `timestep` value and `Context` uses quicksort algorithm (unstable sorting), order between events with the same time step is not defined.
- Hashed map `timesteps` maps each time step to the index of its first occurrence in ordered `calendar`. Intended for fast lookup of events belonging to the same time step.
- `Frame` is hashed map that maps `Node` instances to `Entity` instances. It should be interpreted as location definition for all `Entity` instances at one particular time step. All `Frame` maps are stored in the list `frames`, which is indexed by time steps.
- Color of the `GraphView` scene is saved in `sceneBackground`.
- Information about currently viewed part of `GraphView` scene is saved in `sceneMatrix` and `sceneViewCenter`. Whereas `sceneMatrix` stores transformation matrix for zooming, `sceneViewCenter` stores point in the scene that is aligned to the center of visible area (viewport). To provide backwards compatibility with alpha version of GraphRec, there is also `sceneAngle`, which determines rotation of the scene. However, `sceneAngle` is now *obsolete*, because rotation is applied on graph itself instead of the scene.
- String `validatorName` stores the name of the last validator that validated `Context`.
- Flag `enabledColoring` specifies whether input file has provided explicit coloring information or not. If not, `enabledColoring` is *true* and serves as a hint that implicit colors should be applied.
- Flag `enabledLayouting` specifies whether input file has provided explicit positional information or not. If not, `enabledLayouting` is *true* and serves as a hint that layouting should be applied automatically.
- Intended displacement of nodes is stored in `layoutDisplacement` variable. It serves as a hint for layouters. Note that value is relative and each layouter can interpret it differently.

3.4 Producing Servants

Reference: *servant.h*, *factory.h*, *factory.cpp*

`Servant` is an abstract class, which is intended as a basic interface provided by its implementers. Basic interface includes function `Name()` that returns servant name and function `Description()` that returns its description. Whereas name serves as a unique identification of the servant, description is intended for usage in the GUI (menu entries, status bar labels etc.). Each servant must also provide static version of the name function, called `GetName()`, so that its name can be discovered without instantiation. Both `Name()` and `GetName()` must return the same string.

Classes that implement `Servant` are catalogued in the `Factory` class. Servants are grouped according to their types, which are listed in `ServantType` enumeration. Each type corresponds to abstract class that inherits `Servant` and specifies some additional functions. These specialized interfaces are described in following subsections. When some object needs a servant for some purpose and knows its type and name, it calls `CreateServant()`, which is a `Factory` static function returning the instance of requested servant. `Factory` also provides static function `GetServantNames()` for discovering all available servants of one particular type.

3.4.1 Parser Interface

Reference: *parser.h*

`Parser` is an abstract class, which inherits `Servant` and imposes implementation of two functions:

- `ParseFile()` should search the `file` for graphs, **count statistics** for each found one and insert those data into the table according to provided `header`. Table is composed from root and its children, all of them `TreeWidgetItem` instances. Whereas root only holds file name and file path (as a tool tip), each child stands for one line in the table. Idea is that, when displaying more than one analyzed file to the user, each table can be folded into its root. Collected information includes number of graph elements, solution length, preferred

servants and file location. Order of these items in the table row is specified by parameter `header`, which is a list of entries from `HeaderItem` enumeration. Each `HeaderItem` entry in the list serves as a hint on what information to put in what table column. Function returns reference to the table root. `TreeWidgetItem` inherits `QTreeWidgetItem`, which serves as a basic element for many of Qt data visualization widgets (lists, trees, grids). Only difference between the two is that `TreeWidgetItem` implements differently its comparing function for sorting – it has more universal behavior when comparing various combinations of text and number values.

- `ParseGraph()` should search the `file` on the location specified in the `context`, **completely analyze** graph on this location and insert all data into `context`. It is expected that function builds `nodes`, `entities`, `edges`, `calendar` and fills `graphName`, `enabledColoring`, `enabledLayouting` in the `context`. Optionally it can also fill `validatorName`, `sceneBackground`, `sceneMatrix` and `sceneViewCenter`. Note that `calendar` should be sorted before leaving the function. While parsing the file, it is possible to emit some error messages through `Error()` signal.

Note that `Name()` function of the `Parser` must return string containing information about file suffix in the following format: `*.suffix` (e.g. `MyParser (*.txt)`).

3.4.2 Saver Interface

Reference: *saver.h*

`Saver` is an abstract class, which inherits `Servant` and imposes implementation of three functions:

- `Open()` should open provided `file` and accomplish initialization of the saver. It is also intended for writing file header etc.
- `Save()` should save provided `context` into output file. The structure and amount of data that are going to be saved is entirely up to the `Save()` function.
- `Close()` should safely close output file (if needed). It is also intended for writing file footer etc.

Note that `Name()` function of the `Saver` must return string containing information about file suffix in the following format: `*.suffix` (e.g. `MySaver (*.txt)`).

3.4.3 Validator Interface

Reference: *validator.h*

`Validator` is an abstract class, which inherits `Servant` and imposes implementation of two functions:

- `Validate()` is responsible for exploring `calendar` and building `frames` in the provided `context`. It should set `valid` and `reverse` flags in every `CalendarEvent`. While going through the `calendar`, function should progressively build `Frame` maps from valid movements and insert these maps into `frames` in the `context`. Since `frames` is the main structure needed for animation, `Validate()` is the most responsible function in a matter of what exactly will be animated. While validating the `calendar`, it is possible to emit some error messages through `Error()` signal. Function should also set a `validatorName` in the `context` as a signature.
- `GetColor()` returns some color for every value specified in `ColorScheme` enumeration. Idea behind this function is that all graphs validated by one particular `Validator` should be visually distinguishable from the others. Note that this function serves only as a hint and should be used only if input file did not specified any explicit coloring of graph elements. Since `Validator` interface implements this function itself, its implementation is optional in the implementer.

3.4.4 Layouter Interface

Reference: *layouter.h*

`Layouter` is an abstract class, which inherits `Servant` and imposes implementation of `Layout()` function. It alters positions of `Node` instances contained in `nodes` map in the provided `context`. Positions are set accordingly to implemented layout algorithm. Since some layout algorithms are iterative, it is assumed that function will be repeatedly called by the owner, probably on timer or on separate thread. Consequently, function must return boolean value specifying whether layout is finished (*true*) or not (*false*). Non-iterative algorithm, which calculates layout in a single call, should simply return *true*.

3.4.5 Recorder Interface

Reference: *recorder.h*

`Recorder` is an abstract class, which inherits `Servant` and imposes implementation of `GetSettingsWidget()` function. It should prepare and return `QWidget`, which contains controls connected to the custom slots in the implementer. Returned widget is presented to the user as a part of `CaptureDialog`. When user edits any of these controls in the dialog, all changes are directly sent to the `Recorder` implementer. This mechanism allows the implementer to have almost any specific additional settings that are not covered by interface functions.

3.4.5.1 ImageRecorder Interface

Reference: *imagerecorder.h*

`ImageRecorder` is an abstract class, which inherits `Recorder` and imposes implementation of two functions:

- `GetPaintDevice()` should prepare and return `QPaintDevice` into which the owner will render visual data. Since some Qt classes derived from `QPaintDevice` need additional information for their construction, function takes four arguments – `path` and `name` of the target file (for devices that directly saves the data), `height` and `width` of the image (for raster devices).
- `SaveImage()` should encode the provided `device` to the target file specified by `path` and `name`. After the owner renders data to `QPaintDevice` obtained by `GetPaintDevice()`, it can call `SaveImage()` passing `device` as an argument. Function is intended for devices, which do not save the data directly to persistent storage.

Note: Both functions take target file name as an argument. This name is **incomplete** and serves only as a template. Function should search the destination for the name conflicts and appropriately edit provided file name to be unique (e.g. by adding number). Format suffix must be also appended to the name.

3.4.5.2 VideoRecorder Interface

Reference: *videorecorder.h*

`VideoRecorder` is an abstract class, which inherits `Recorder` and imposes implementation of three functions:

- `GetFPS()` returns the intended frame rate for the video stream.
- `Start()` function should initialize the encoder and then run a consumer thread, which will encode `QImage` instances from the global buffer `G_GRVideoBuffer` of the size `G_GRVideoBufferSize`. Function takes four arguments – `path` and `name` of the target file, `height` and `width` of the video frame. The `name` is incomplete and serves only as a template. Function should search the destination for the name conflicts and appropriately edit provided file `name` to be unique (e.g. by adding number). Format suffix must be also appended to the `name`. Access to the global buffer is guarded by two semaphores - `G_GRVideoFree` and `G_GRVideoUsed`. All these global variables with `G_GR` prefix are declared in the `main.cpp` file and can be accessed by only one producer and one consumer at a time. Since GraphRec architecture allows the user to request more than one video encoding at a time, there is a danger of having more than one concurrent consumer. It would certainly lead to the corrupted video. Thus, before doing anything with those global

variables, `Start()` function have to check whether `G_GRVideoOwner` pointer is null and does not point to some other `VideoEncoder` implementer. In the case that the pointer is null, it should be initialized by self-reference of the implementer. This approach guarantees exclusive access to the global variables. Note that the whole idea behind producer/consumer synchronization and video encoding is described in other sections of this manual.
- `Stop()` function should wait until `G_GRVideoBuffer` is emptied and should safely terminate the consumer thread. After that, the output file should be ended (probably with some footer or trailer) and closed. Before function returns, the `G_GRVideoOwner` pointer has to be set to null value.

3.5 GraphView Class

Reference: *graphview.h*, *graphview.cpp*

`GraphView` class inherits `QGraphicsView`, which is a class for advanced two-dimensional graphics. `GraphView` extends its ancestor by several custom-made functions and usage of additional dialogs, controls and servants. Since `GraphView` is quite complex class, its description is divided into several subsections. This introductory section will describe only `GraphView` construction and its connections with other classes.

Usually, it is not very useful to describe class constructor in the documentation. However, in the case of `GraphView` constructor, systematic description very well supports the overall idea over the low-level application structure and data flow:

1) Arguments passed to the constructor consists of graph location (`file`, `filePosition`) and servant names (`parserName`, `validatorName`, `layouterName`).
2) `QGraphicsScene` is created and initialized by calling inherited `setScene()` function. Scene is the core part of every `QGraphicsView`. For more information about `QGraphicsView`, `QGraphicsScene` and its coordinate system, please refer to the Qt documentation.
3) `Context` (further accessible through `m_context`) is created and information about graph location (`file`, `filePosition`) is stored into it.
4) `ErrorDialog` (further accessible through `m_dialogError`) is created.
5) `Parser` corresponding to `parserName` is obtained from `Factory` and its `Error()` signal is connected to `Log()` slot in the `ErrorDialog`.
6) Input `file` is parsed by `Parser::ParseGraph()` on specified `filePosition`, effectively filling almost all information in the `m_context`.
7) Scene background, scene matrix and viewport central point are set according to `m_context`.
8) `Validator` corresponding to `validatorName` is obtained from `Factory` and its `Error()` signal is connected to `Log()` slot in the `ErrorDialog`.
9) Structures in the `m_context` are validated, which in fact constructs `m_context->frames`. If the input file did not specified coloring, default colors of chosen `Validator` are injected into graph elements in the `m_context`. Otherwise, coloring has been already done by parser.
10) All instances of `QGraphicsItem` from the `m_context` (that is `Node` and `Edge` instances) are added into the scene.
11) `Layouter` corresponding to `layouterName` is obtained from `Factory` and saved into `m_layouter`.
12) `QTimer` for layouting (further accessible through `m_timerLayout`) is created and connected to `on_timerLayout_timeout()` handler.
13) If the input file did not specified positioning, layout is firstly randomized by `LayoutRandomize()` and then properly calculated by `m_layouter`.
14) All nodes are connected to `GraphView` signals and slots, which are further utilized to distribute several parameters into and from them during user actions and during animation. Especially note the `NodeMoved()` slot, which notifies `GraphView` about layout changes.
15) `QTimeLine` for animation (further accessible through `m_animationTimeLine`) is created and connected to `AnimateTimeStep()` slot.
16) Signal `AnimationStepDone()` is connected to `AnimateTimeStep()` slot.

17) Additional visual controls are created - two `QSlider` instances (`m_sliderTimeStep`, `m_sliderDisplacement`) and one `QSpinBox` instance (`m_spinBoxTimeStep`). Both controls referring to time step are connected with each other to reflect the same information all the time. All three controls are connected to `GraphView` slots (`SetTimeStep()` or `SetDisplacement()`). Note also that their focus is forwarded to the `GraphView` (this is very important, because user expects `GraphView` to connect to `GraphRec` by clicking anywhere into its area).
18) Widgets created in the last step are placed onto `GraphView` providing quick actions to the user.
19) `ColorDialog` (further accessible through `m_dialogColor`) is created.
20) Settings are fetched from persistent storage.

Following list classifies output connections (signals) from `GraphView` to other classes:

- `GraphRec` as an owner: `HasFocus()`, `ValidatorNameChanged()`, `LayouterNameChanged()`, `ValidatorDescriptionChanged()`, `LayouterDescriptionChanged()`, `Message()`, `DiscreteDisplacementEnabled()`, `LayoutingEnabled()`, `NodeLabelsChanged()`, `LayoutingInProgress()`, `AnimationInProgress()`. Note that all these signals can be fired at once by calling `UpdateConnections()`.
- All `Node` instances: `TimeStepChanged()`, `DiscreteDisplacementEnabled()`, `DiscreteDisplacementOffsetChanged()`.
- Additional controls (`m_sliderTimeStep`, `m_sliderDisplacement`, `m_spinBoxTimeStep`, `m_sliderDuration`): `TimeStepChanged()`, `DisplacementChanged()`, `DurationChanged()`.
- `GraphView` itself: `AnimationStepDone()`.

3.5.1 Error Dialog

Reference: *errordialog.h*, *errordialog.cpp*, *errordialog.ui*

`ErrorDialog` is a class, which inherits `QDialog` and has its GUI part accessible through `m_ui` pointer. Class provides public `Log()` slot, which can be used by other classes to post error messages into `QListWidget` (`m_ui->listWidgetErrorLog`) in the `ErrorDialog`. Whenever user selects some error message (`QListWidgetItem`), it is analyzed by regular expression `m_errorRegExp`. If the expression matches, `ErrorDialog` emits `ErrorSelected()` signal, which is connected to `GraphView::SelectEvent()` slot. Selected event is then tracked in the `GraphView::m_context`. If the event exists, its time step is set and corresponding nodes are highlighted. Whole log can be also saved into simple text file (`on_buttonSave_clicked()`). `ErrorDialog` is non-modal and is constructed in the `GraphView` constructor and destroyed in its destructor. During the lifetime of the owner, `ErrorDialog` can be only shown or hidden through `GraphView::m_dialogError`.

3.5.2 Color Dialog

Reference: *colordialog.h*, *colordialog.cpp*, *colordialog.ui*

`ColorDialog` is a class, which inherits `QDialog` and has its GUI part accessible through `m_ui` pointer. GUI is represented by `QTabWidget` with three tabs, each containing one `QTreeWidget`. Order of tabs is inferred from `TabOrder` enumeration. Header of the first tree widget is constructed according to `Node::NodeColorType` enumeration. Similarly, header of the second tree widget corresponds to `Entity::EntityColorType` enumeration. For the fast determination of both color types, column numbers are mapped to them in the `nodeColorTypes` and `entityColorTypes` maps. Whereas first and second tree widget is intended for listing nodes and entities (both obtained from `m_context` argument passed to the constructor), third tree widget contains more general items – namely `m_itemBackground`, `m_itemBoundary` and `m_itemHighlight`. Column order and their captions can be easily changed in the constructor.

`ColorDialog` is designed to be non-modal, effectively allowing selection of nodes or entities directly from `GraphView`. Whereas `GraphView::keyPressEvent()` on *Ctrl* key enables nodes selection by mouse dragging, `GraphView::keyReleaseEvent()` on *Ctrl* key appends all selected nodes to the list and passes it to the `ColorDialog::SelectItems()` function, which selects

corresponding items in the tree widget (either nodes or entities, depending on what tree widget is currently active).

Color changes are done in a flexible but rather complicated way. Every time the tab is changed, `on_tabWidget_currentChanged()` calls `CreateMenu()` function, which constructs customized menu for current tree widget. Menu actions correspond to tree widget columns and are connected through `QSignalMapper` to the `SetColor()` slot. Menu is then embedded into `m_ui->buttonSetColor` and serves also as a tree widget context menu displayed by `ShowContextMenu()` function. Whenever user selects one or more items in the tree widget and hits some menu action, `SetColor()` is called with column number obtained from the signal mapper. Color palette is then showed (static function `QColorDialog::getColor()`) and after the user chooses appropriate color, it is saved for all selected items into the `m_context`, each time inferring color type from `nodeColorTypes` or `entityColorTypes`. Note that `m_itemBoundary` and `m_itemHighlight` must be treated in a special way, since it affects more than one graph element in the `m_context`. Another exception is `m_itemBackground`, which is connected to `GraphView` through `BackgroundColorChanged()` signal.

`ColorDialog` is constructed in the `GraphView` constructor and destroyed in its destructor. During the lifetime of the owner, `ColorDialog` can be only shown or hidden through `GraphView::m_dialogColor`. Since it is not possible to change colors elsewhere, `ColorDialog` contains actual color information for all graph elements at any time.

3.5.3 Layouting

Reference: *graphview.h*, *graphview.cpp*, *node.cpp*, *layouter.h*

Mechanism of automatic continuous layouting is based on the ticking of `m_timerLayout` and its handler `on_timerLayout_timeout()`, which calls repeatedly `m_layouter->Layout()` until the false value is returned informing that layout is finished. Automatic continuous layouting is enabled by calling `SetLayoutingEnabled()`, which sets the `m_context->enabledLayouting` variable. By calling `SetDisplacement()`, it is possible to change `m_context->layoutDisplacement`, which is fetched by `Layouter::Layout()` from the `m_context`. Timer `m_timerLayout` can be controlled by `LayoutStart()` and `LayoutStop()` functions. Note that `m_timerLayout` can be also launched indirectly by moving any node while automatic layouting is enabled – it signals `NodeMoved()` slot, which calls `LayoutStart()`.

Discrete layouting is done differently. It can be enabled by calling `SetDiscreteDisplacementEnabled()`, which sets the `m_discreteDisplacementEnabled` variable. Function `drawBackground()` then repeatedly paints grid consisting of horizontal and vertical lines on the background of the `GraphView` scene. Color of the lines is inversed RGB value of the background color. Offset between lines in the grid can be set by calling `SetDiscreteDisplacementOffset()`, which sets `m_discreteDisplacementOffset`. Both `m_discreteDisplacementEnabled` and `m_discreteDisplacementOffset` are sent to all nodes via `DiscreteDisplacementEnabled()` and `DiscreteDisplacementOffsetChanged()` signals whenever changed. Every node then calls `Node::AlignPoint()` each time its position is altered (`Node::itemChange()`), effectively snapping itself to the closest intersection of horizontal and vertical line. All nodes can be snapped to the grid at once by calling `LayoutDiscrete()`.

It should be noted that node positions are locked during animation by `LayoutLock()` function, because it would otherwise lead to entities visually missing their destinations. When both `m_context->enabledLayouting` and `m_discreteDisplacementEnabled` are disabled, nodes can be freely moved by the user. When needed, layout can be randomized by `LayoutRandomize()` function.

3.5.4 Scene Actions

Reference: *graphview.h*, *graphview.cpp*

All functions described in this section are usually called from `keyPressEvent()` handler. Graph can be moved by calling `ScrollGraph()`, which in fact move every node from `m_context` by calling

`Node::moveBy()`. Graph moving is only possible when layouter is not working, because it might destabilize its algorithm. Note that viewport scrolling, which is handled by `QGraphicsView` itself, is still possible by mouse dragging even when layouting is in progress. Zooming is done by `ScaleView()`. Provided scaling factor is first tried on the scene matrix in order to prevent very large/small zoom, and then applied by calling `scale()` function inherited from `QGraphicsView`. `ScaleView()` is usually called by `whellEvent()` handler, which calculates scaling factor from mouse wheel rotation.

The most complicated action is rotation. Its implementation can be found in `RotateGraph()` function. It should be noted that alpha version of GraphRec rotated the whole scene. Since this approach lead to inconsistencies with other features, graph is currently rotated by changing positions of its nodes instead. This also implies that `Context::sceneAngle` is no longer needed and only kept for backwards compatibility with files created by alpha version (such files are now transformed in the `GraphView` constructor to be compatible with new approach). Another limitation, similarly to scrolling, is that graph can be rotated only when layouter is not working due to destabilization of its algorithm. Discrete positioning is also ignored during rotation, because it is in conflict with implemented rotation mechanism. Rotation itself is done in the following way. All nodes are grouped into the `QGraphicsItemGroup` and current cursor position is determined by `QCursor::pos()`. Then, `QTransform` is applied onto the group by virtually moving the whole group to the cursor position, rotating it here by the given angle and moving it back to its original position. This ensures that graph is rotated over the cursor position, which is more flexible than rotation over viewport center or even graph center (which is expensive to calculate).

3.5.5 Animation

Reference: *graphview.h*, *graphview.cpp*

Usually, before the animation is started, user sets the initial time step for animation by calling `SetTimeStep()`. It simply takes all nodes and updates their entities according to `m_context->frames`. In order to do that, `GraphView` stores information about current time position in `m_calendarPosition` and `m_timestep` – both of which are kept synchronized with help of `m_context->timesteps`. Whenever `m_timestep` variable is altered by some function, the function also emits `TimeStepChanged()` in order to update time step in all nodes, so that nodes know whether they are in the final position or not. From the hindsight, animation is a sequence of `AnimateTimeStep()` calls. Every `AnimateTimeStep()` in the sequence either starts `m_animationTimeLine` for visible animation or emits `AnimationStepDone()` for fast skip. Both signals, `m_animationTimeLine->finished()` and `AnimationStepDone()`, are again connected to `AnimateTimeStep()`, effectively acting as endless loop. Loop can be controlled by `AnimationStart()` and `AnimationStop()` functions. `AnimationStart()` sets the `m_animationIsRunning` flag, so that other functions can discover whether animation is in progress, and calls `AnimateTimeStep()` to start the loop. `AnimationStop()` only sets `m_animationStopRequest` flag, which is periodically checked by `AnimateTimeStep()`. There is also function `AnimationStep()`, which is only simple subsequent call of `AnimationStart()` and `AnimationStop()`. Considering animation, there are two layouting problems that deserve special attention. First problem is that animation assumes static positions of graph elements – animated movements are precalculated for every time step. Thus, layouting must be paused during animation, because animated entities would otherwise miss their possibly moving destinations. Second problem is that user can change layouting settings during animation. In order to react correctly to these situations, `AnimationStart()` lock node positions (`LayoutLock()`) and sets pair of flags - `m_layoutingWasEnabled` (automatic continuous layouting mode was enabled at the moment) and `m_layoutingWasRunning` (layouter was still working at the moment).

`AnimateTimeStep()` itself is quite large function, which is composed from several logical parts (recommended reading order is 3, 4, 1, 2):

1) Processing data structures from the last call. All shallow copies from the `m_nodeBuffer` are removed from the scene and destroyed; their entities are inserted into corresponding destination nodes (also discovered from `m_nodeBuffer`). Shallow copies from `m_edgeBuffer` are also removed from the scene and destroyed. Finally, animations from

the m_animationBuffer are destroyed, leaving all three data structures empty for the next round.

2) While deciding whether to stop animation loop, m_animationStopRequest is checked. If it is true, m_animationIsRunning flag is reset and node positions are unlocked by LayoutLock(). Note that there are two cases, in which the layouter should be immediately launched. First case occurs when the layouter was running just right before the animation and automatic layouting is still enabled now (means that the layout is unfinished). Second case occurs when automatic layouting was not enabled before animation but is enabled now. Both cases can be inferred from m_context->enabledLayouting, m_layoutingWasEnabled and m_layoutingWasRunning flags.
3) Preparing data structures for the animation and the next call. Every valid CalendarEvent structure with the same time step as m_timestep is fetched from m_context->calendar. Shallow copy of the event source node is inserted into the scene and its QGraphicsItemAnimation is created, connected to m_animationTimeLine and appended to m_animationBuffer list. Each QGraphicsItemAnimation has defined its starting and ending point, which correspond to the position of associated source and destination node. Idea is that m_animationTimeLine acts as a director, who sets time in all connected QGraphicsItemAnimation instances, which move their embedded QGraphicsItem along the predefined pathway. Entity is removed from the source node immediately after the shallow copy is created, effectively leaving the copy as the entity carrier. Both shallow copy and destination node are always appended as a pair into the m_nodeBuffer. Note that shallow copy is not connected to any of GraphView signals. Thus, it is safe to increment m_timestep even while the entities are not yet in their destinations. Edge between the event source node and destination node is also shallow copied and the copy is set to the highlighted mode. All such edges are appended to m_edgeBuffer.
4) Depending whether just animating or capturing (m_recordingEnabled), m_animationTimeLine->start() is called or AnimationStepDone() is emitted. Signal is emitted also in the case of empty m_animationBuffer or zero duration of m_animationLine (inferred from m_timeoutZero flag), because animation would be invisible and only slowing down the process.

3.5.6 Setup Dialog

Reference: *setupdialog.h*, *setupdialog.cpp*, *setupdialog.ui*

SetupDialog is a class, which inherits QDialog and has its GUI part accessible through m_ui pointer. SetupDialog is implemented in a straightforward way providing get and set functions for almost all of its control widgets. Dialog is utilized by GraphView::ShowSetupDialog() – firstly inserting GraphView variables into SetupDialog controls, then executing it as a modal dialog and finally fetching values back to GraphView variables. The only interesting thing about dialog implementation is that change signals of its control widgets are connected to the ChangeTrigger() slot, which emits Changed() signal that is connected to m_ui->buttonDefault enabler. Thus, if the settings are changed and user hits the enabled button, they are saved as default global settings for any subsequent GraphView constructor.

3.5.7 Capture Dialog

Reference: *capturedialog.h*, *capturedialog.cpp*, *capturedialog.ui, graphview.h*, *graphview.cpp*

CaptureDialog can be invoked from the main menu through two different actions, which are connected to corresponding slots in the GraphView – Snapshot() or Sequence(). Both slots call GraphView::ShowCaptureDialog() with corresponding CaptureDialog::Mode as a parameter. ShowCaptureDialog() closes and destroys existing GraphView::m_dialogCapture and constructs new one again with corresponding CaptureDialog::Mode. After the dialog is constructed, settings are injected into it, its signals for accepting and rejecting are connected to GraphView::CaptureDialogHandler() and finally, depending on the passed mode, it is invoked as a modal (mSequence mode) or non-modal (mSnapshot mode) dialog.

CaptureDialog is a class, which inherits QDialog and has its GUI part accessible through m_ui pointer. Most of capture settings provided by dialog are implemented in a straightforward get/set way. Interesting is only m_ui->comboBoxRecorder containing list of available recorders (discovered

from `Factory`). Every time some recorder is selected (`on_comboBoxRecorder_currentIndexChanged()`), its instance is retrieved from `Factory` and saved into `m_recorder` pointer. Since `Recorder` interface defines `Recorder::GetSettingsWidget()` function, it is called and returned `QWidget` is saved into `m_widgetRecorder` and embedded in the `CaptureDialog` window. Behavior of the `CaptureDialog` depends on `Mode` passed to the constructor. First of all, some GUI controls are disabled in either mode (in more detail described in the *User's Guide*). More importantly, whereas `mSnapshot` mode only allows listing of recorders that implement `ImageRecorder`, `mSequence` allows all available recorders. Different is also dialog confirmation handler (`on_buttonCapture_clicked()`), which in `mSnapshot` mode only emits signal and leaves dialog opened.

Back in `GraphView`, `CaptureDialogHandler()` is called as a reaction on `CaptureDialog` confirmation. Depending on whether dialog was accepted or not, settings are then fetched from it into `GraphView` private variables and selected recorder is saved to `GraphView::m_recorder` pointer. Further solving of the task is leaved for either `GraphView::SnapshotHandler()` or `GraphView::SequenceHandler()`, both of which are described in the next section.

3.5.8 Rendering

Reference: *graphview.h*, *graphview.cpp*, *main.cp*, *imagerecorder.h*, *videorecorder.h*

Rendering is covered by `Render()` function. For now, let us only say that function behaves differently on what `GraphView::RenderMode` is stored in the `m_renderMode` variable – either directly saving image to persistent storage (`rmDirectSave` mode) or inserting image into `G_GRVideoBuffer` (`rmBufferSave` mode). `Render()` function might be invoked from four places in the source code. These entry points represent four different approaches how rendering can be handled:

1) `SnapshotHandler()` sets `m_renderMode` to `rmDirectSave` and simply calls `Render()`, which renders and saves **single image**.
2) `SequenceHandler()` infers that `m_recorder` is `ImageRecorder`. Thus, `m_renderMode` is set to `rmDirectSave`, because expected result is sequence of images. Scene is rendered at every `m_captureInterval` between two corresponding time steps (when nothing is moving), both of which are bounded by `m_captureTimeStepBegin` and `m_captureTimeStepEnd`. `SequenceHandler()` further decides depending on `m_recordingInteractive` variable:
    a) **Non-interactive image sequence capturing** is handled directly by `SequenceHandler()`. Time steps are cycled by `SetTimeStep()` in the loop, whose each iteration calls `Render()` function. In order to improve performance and responsiveness of the application, scene is not rendered, simple `QProgressDialog` is shown (`m_dialogProgress`) instead and application message loop is emptied during each iteration.
    b) **Interactive image sequence capturing** is handled by `AnimateTimeStep()` function. `SequenceHandler()` only sets initial time step, toggles `m_recordingEnabled` flag and calls `AnimationStart()`. `AnimateTimeStep()` will be normally preparing and animating time steps, each time checking for correct combination of `m_recordingEnabled` and `m_renderMode`, which allows calling `Render()`.
3) `SequenceHandler()` infers that `m_recorder` is `VideoRecorder`. Thus, `m_renderMode` is set to `rmBufferSave`, because expected result is **video file**. Firstly, frame rate is retrieved from `m_recorder` by calling `VideoRecorder::GetFPS()` and saved into `m_fps` variable. Then, `m_recorder` is started by calling `VideoRecorder::Start()`, which takes destination (`m_captureFilePath`, `m_captureFileName`) and resolution (`m_captureWidth`, `m_captureHeight`) as arguments. From now on, `m_recorder` acts as a consumer, who progressively fetch visual data from `G_GRVideoBuffer` and saves it into the file. Producer will be `Render()` function together with `AnimateTimeStep()`, which is immediately invoked by calling `AnimationStart()`. Note that difference between interactive and non-interactive capturing (`m_recordingInteractive`) is not so big in comparison with image sequences – it is now handled by the same code in the `AnimateTimeStep()` and the only difference is (in)visible scene and utilization of `m_dialogProgress`. Rendering in `AnimateTimeStep()` is

done efficiently – after checking correct combination of `m_recordingEnabled` and `m_renderMode`, nodes are animated only at positions required by video frame rate. Thus, instead of calling `m_animationTimeLine->start()`, position of every `QGraphicsItemAnimation` from `m_animationBuffer` is explicitly set in the loop that iterates as many times as there are frames that fit, according to `m_fps`, into `m_animationTimeLine->duration()`. Note that position is still inferred from `m_animationTimeLine`, because it might not be linearly dependent on time. Every loop iteration calls `Render()` and empties application message loop due to responsiveness. After the loop is finished, `AnimationStepDone()` signal is emitted. When all time steps are captured, `AnimationStop()` function calls `VideoRecorder::Stop()` on `m_recorder` in order to safely finish recording. This approach ensures that video is rendered as fast as possible and the quality of output is not dependent on processor speed (e.g. dropped frames). However, this also implies that, due to processor speed, interactive video capturing is not real time – from the user's point of view, it might be either extremely slow or extremely fast, both of which are not ideal for interactivity.

`Render()` function can be described in three steps:

- **Preparing** `QPainter`:
  - In `rmDirectSave` mode, painter is constructed from `QPaintDevice`, which is retrieved from `m_recorder` by calling `ImageRecorder::GetPaintDevice()`. Note that, apart from `m_captureWidth` and `m_captureHeight`, which are quite logic, function takes also `m_captureFilePath` and `m_captureFileName` as arguments, because some paint devices directly save data to persistent storage while rendering (e.g. XML file in the case of SVG file format).
  - In `rmBufferSave` mode, painter is constructed from `QImage`, which itself is constructed according to `m_captureWidth` and `m_captureHeight`.
- **Rendering** image. Since viewport of `GraphView` might have different aspect ratio than the one calculated from `m_captureWidth` and `m_captureHeight`, rectangle representing exposed area of the scene must be appropriately extended. At first, rectangle is aligned to the top left corner of the viewport and then either its width or height is increased to match output ratio. This ensures that resulting image will certainly contain intended part of the scene and will have correct aspect ratio. Finally, exposed area is rendered by the painter into its embedded device.
- **Saving** `QPaintDevice`:
  - In `rmDirectSave` mode, device is saved by passing it to `ImageRecorder::SaveImage()`. Note that, this function might do nothing since data might have been already saved by device itself.
  - In `rmBufferSave` mode, device, which is in fact `QImage`, is saved into `G_GRVideoBuffer`, which is guarded by two semaphores - `G_GRVideoFree` and `G_GRVideoUsed` – in a manner of producer/consumer synchronization. Current buffer position is stored in `m_bufferPosition` variable. Note that acquiring the semaphore is regularly interrupted by emptying the application message loop due to responsiveness.

3.5.9 Video Encoding

Reference: *videorecorder.h*, *ffmpegvideorecorder.h*, *ffmpegvideorecorder.cpp*

Video encoding is implemented in `FFmpegVideoRecoder` class, which inherits `VideoRecorder` interface. Class embeds `EncoderThread` derived from `QThread`, which is intended as a consumer for `G_GRVideoBuffer`. Moreover, class is heavily dependent on API functions of FFmpeg video library. In order to manage and distribute data in a uniform way among these API functions and embedded thread, class defines structure `Data`, which contains all required parameters and FFmpeg data structures. `Data` structure, created in the `FFmpegVideoRecoder` constructor, is accessible through `m_data` or `EncoderThread::m_recdata` pointer (both classes share the structure, but `FFmpegVideoRecoder` is the owner responsible for deletion). Following description focuses on how the encoding is handled in detail. In order to provide clear explanation, API calls are, in most cases, not mentioned explicitly. It should be noted that due to the lack of proper FFmpeg

documentation, API calls are deduced from `output-example.c` provided in FFmpeg source code package.

Let us assume that `Data` structure already contains some initialized entries, which have been set by slots connected to widget provided to the `CaptureDialog`:

- **Initialization** of `Data` structure is further done by `Start()` function:
  a) File suffix and video format are inferred from `m_data->formatString`. Both `m_data->format` and `m_data->context` structures are initialized by FFmpeg API calls. Since FFmpeg guesses format from given string, it is easy to add more video formats if needed.
  b) Structure `m_data->stream` is initialized by calling `CreateStream()` function, which further calls API. Note that the `CreateStream()` also contains all codec settings, some of which are hard coded.
  c) By a series of API calls, `OpenVideo()` function opens codec, whose settings were just set in `CreateStream()`. After that, encoding buffer is allocated (`m_data->buffer`, `m_data->bufferSize`). Finally, a pair of `AVFrame` structures is initialized by `AllocateFrame()` function – whereas format of `m_data->frameTemp` must be compatible with `QImage` (RGB), `m_data->frameFinal` is intended as a codec input (where the most suitable format is YUV).
  d) Output file is created and opened (`Start()` takes path and name as arguments). Format header is written into the file by API call.
  e) Encoding thread is invoked by calling `m_thread->start()`, which effectively starts `EncoderThread::run()` on the different thread.
- **Encoding** of images from `G_GRVideoBuffer` is done by `EncoderThread::run()` function running on worker thread:
  a) First of all, relevant data are copied from `EncoderThread::m_recdata` into local variables of `EncoderThread::run()` function. Thus, data (mainly pointers to FFmpeg structures) are now located on the local stack and one level of indirection is avoided. Note that this is **not** intended as a protection against race conditions – FFmpeg structures are still accessible from both threads since only pointers are copied. Moreover, copying is not guarded by any mutex. None of this is problem, because worker thread has exclusive access assuming current architecture (main thread only prepares those structures before starting worker thread – there is no concurrency between them concerning mentioned data).
  b) Function enters the infinite loop, which is encoding images from `G_GRVideoBuffer`, until the buffer is empty and `EncoderThread::m_terminate` flag is toggled. Since anything done in the loop has direct impact on encoding speed, it must be implemented efficiently. Every `QImage` (RGB) is at first copied from `G_GRVideoBuffer` into the local `frameTemp` (RGB). Note that copying is the only thing guarded by semaphores. Thus main thread, as a producer, is not slowed down by waiting on actual frame encoding.
  c) Visual data in the `frameTemp` are converted from RGB to YUV format and saved into `frameFinal`. Highly optimized algorithm for this conversion is implemented in the `swscale()` function provided by GPL version of FFmpeg (as of 2009, LGPL version of `swscale()` is planned).
  d) Image stored in `frameFinal` is encoded by chosen codec and saved into output file (both actions done by API calls). Note that encoding function utilizes `buffer`, whose `bufferSize` is set to 8MB by default (might be changed by user through GUI). If the `buffer` is too small, recording immediately fails or video will be corrupted.
- **Deinitialization** and memory freeing is done by `Stop()` function:
  a) `EncoderThread::SafelyTerminate()` is called, which effectively terminates worker thread by setting its `m_terminate` flag.
  b) Format trailer is written into the output file by API call.
  c) `CloseVideo()` and `DestroyStream()` deletes all allocated FFmpeg structures.
  d) Output file is closed.

3.6 Main Window

Reference: *graphrec.h*, *graphrec.cpp*, *graphrec.ui*, *graphview.h*, *opendialog.h*

`GraphRec` is a class, which inherits `QMainWindow` and has its GUI part accessible through `m_ui` pointer. Class represents the main application window consisting of menu bar (`m_ui->menuBar`), tool bar (`m_ui->toolBar`), status bar (`m_ui->statusBar`) and laid out central widget (`m_ui->centralWidget` or `m_ui->gridLayout`). Menu bar is composed of several submenus and actions (`QAction`). Menu structure and a brief description of actions can be found in *User's Guide*. Tool bar provides a subset of frequently used menu actions. Note that tool bar is designed by programmer and cannot be edited by the user at runtime. During runtime, it is only possible to turn the tool bar on/off or dock it to various sides of the window. Status bar contains two labels – one for displaying application status (`m_labelStatus`) and the other for displaying validator name (`m_labelValidator`). Status bar also allows posting some temporary messages over the labels.

Initially, the central widget is empty and almost all parts of the window and menu are disabled. When user clicks on the *File – Open* button, function `on_actionOpen_triggered()` is called. At first, it invokes `OpenDialog`, which is described in its own subsection. However, when the dialog is confirmed by the user, function searches its `acceptedSolutions` (list of `OpenDialog::SolutionInfo` structures) and process them one after the other. Each `SolutionInfo` provides file name, file location and names of preferred parser, validator and layouter. Function opens the file and creates new `GraphView` by providing it opened file and information from `SolutionInfo`. Created `GraphView` is then added as a tab into either existing or newly constructed `QTabWidget` (depending on whether there is already one). Newly constructed `QTabWidget` is appended into `m_tabWidgets` list and embedded into central widget of the window. Function `ResetControls()` is then called in order to enable/disable menu actions. When user closes the tab (`TabCloseRequested()`), its `GraphView` is destroyed and tab is removed from its tab widget. If the tab widget has no more tabs it is also destroyed and removed from both `m_tabWidgets` and central widget. Function `ResetControls()` is called again.

Previous paragraph hints that each tab widget can contain more than one `GraphView` instance. However, it is even more complicated, because tab widget can be split into more tab widgets. Splitting tab widgets is described in its own subsection. Anyway, since majority of menu actions are only handles, which further calls functions in `GraphView`, it is clear that there must be some mechanism for delivering these calls to the right `GraphView` instance. Note that this mechanism must deliver calls also in the opposite direction (from `GraphView` to `GraphRec`) in order to update labels and some menu selections. Whole problem is resolved by usage of signal/slot mechanism. Many of `GraphRec` actions or signals are connected to `GraphView` slots and vice versa. Every time the `GraphView` is changed (it can be done by either tab change or focus change), `FocusedGraphViewChanged()` is called. Function disconnects `GraphRec` from `m_currentGraphView` and then reconnects it to the `GraphView` provided in an argument `who`. After reconnection is done, connected `GraphView` is requested to emit its status by calling its function `UpdateConnections()`. In order to make all this working, every `GraphView` has its `HasFocus()` signal connected to the `GraphRec` immediately after the construction.

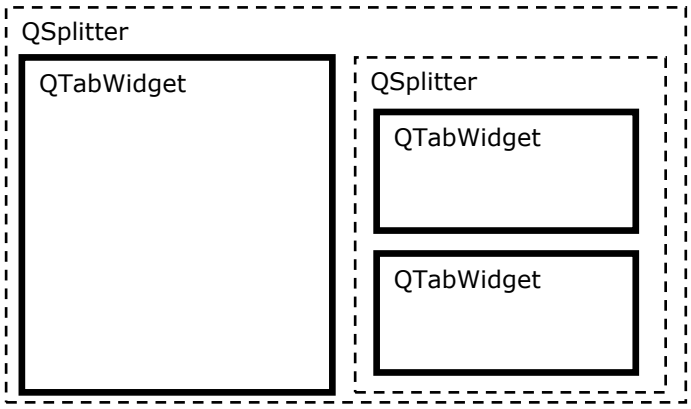
`GraphRec` class is also partially responsible for file saving. When user clicks on *File – Save* or *File – Save All* button, function `on_actionSave_triggered()` or `on_actionSaveAll_triggered()` is called. Both functions invoke `QFileDialog` filling its file format combo box with names of `Saver` servants obtained from `Factory`. After the user chooses path, name and suffix, file is opened and corresponding saver is built. At first, saver is initialized by `Saver::Open()`. Then, it is then passed to the `SaveContext()` function of the currently connected `GraphView` (or to every `GraphView` in the case of *Save All* action). In the end, file is terminated by `Saver::Close()` and closed.

3.6.1 Splitting

Reference: *graphrec.cpp*

Central area of main window is represented by `m_ui->centralWidget` with `m_ui->gridLayout` applied on it. In a basic case, there is only one instance of `QTabWidget`, whose parent is just mentioned `m_ui->gridLayout`. Let us say that the tab widget contains more than one tab (each

tab containing instance of `GraphView`) and currently selected tab is not the first one (left most). Then, it is possible to split this tab widget by `Split()` function into two tab widgets – first containing tabs up to (not including) the selected tab, second containing the rest. Original tab widget is then replaced by instance of `QSplitter` (supports movable boundary between its children) into which those two new instances of `QTabWidget` are added. Described process can be repeated on the arbitrary level of the hierarchy represented by binary tree, whose inner vertices are instances of `QSplitter` and leafs are instances of `QTabWidget`. Tab widget to be split is inferred from parent pointer of currently focused tab (`m_currentGraphView`). Every split can be done either horizontally or vertically.



There is also inverse operation to the splitting. `Unsplit()` function takes `QSplitter` as an argument and, by recursively calling itself on the splitter's children, returns `QTabWidget` containing all tabs from accessible leafs. If user unsplits a tab widget by calling `on_actionUnsplit_triggered()`, its parent `QSplitter` is passed to `Unsplit()` function, effectively merging focused tab widget with its unambiguous neighbour branch (vertical or horizontal). Resulting tab widget is put on the place where destroyed parent splitter originally was. As for tab closing, if the tab is closed and there are no more tabs in the tab widget, it is destroyed and its parent splitter is replaced by the sibling (either `QSplitter` or `QTabWidget`). Note that both `Split()` and `Unsplit()` are maintaining current set of tab widgets in `m_tabWidgets` list.

In the situation, when there is more than one tab visible to the user, it is possible to run animation on all of them at once. As was explained in the animation description, every animation is independently timed by its own `QTimeLine`, which behaves similarly to `QTimer`. Since timers are not precise on the non-realtime operating system, it is difficult to synchronize more of them running in parallel. This implies that two parallel animations with the same duration might get desynchronized after a few time steps. To prevent this from happening, user can toggle *Animation – Synchronize All* action, which modifies behavior of some functions. When synchronization is toggled, `on_actionPlayAll_triggered()` function adds all foreground `GraphView` instances into `m_synchronizedGraphViews` and instead of calling `GraphView::AnimationStart()`, it calls `GraphView::AnimationStep` on all of them. After creation in `on_actionOpen_triggered()`, every `GraphView` had its `Stepped()` signal connected to `GraphRec::SynchronizeAll()` slot. Thus, when any `GraphView` finishes animation of the time step, `SynchronizeAll()` is called. It evaluates whether all members of `m_synchronizedGraphViews` have already finished their step and if so, it calls `GraphView::AnimationStep` on all of them to continue the animation. Note that in the case of different durations among animations, synchronization always waits for the slowest `GraphView` leaving others stopped for a while. In order to assure consistent behavior with other features, `m_synchronizedGraphViews` must be managed by other menu actions related to animation (mainly removing of some `GraphView` instances from the list).

3.6.2 Open Dialog

Reference: *opendialog.h*, *opendialog.cpp*, *opendialog.ui*, *parser.h*

`OpenDialog` is a class, which inherits `QDialog` and has its GUI part accessible through `m_ui` pointer. GUI is represented by two `QTreeWidget` instances (`m_ui->listFound`, `m_ui->listChosen`) and

some buttons. Both tables have its header derived from `headerTemplate`, a list of `HeaderItem` values from `Parser` interface, specified in the `OpenDialog` constructor. Note that table columns containing important but not interesting information for the user are hidden by `HideColumn()` function.

Handler of *Add files* button (`on_buttonAddFiles_clicked()`) invokes `QFileDialog` filling its file format combo box with names of `Parser` servants obtained from `Factory`. After the user chooses appropriate format and files to open, corresponding `Parser` is built and its `ParseFile()` function is called for every selected file. Function call returns `QTreeWidgetItem` acting as a root for multiple other `QTreeWidgetItem` instances each of which represents one table row showing statistics for one graph. Since `headerTemplate` is also passed to the `ParseFile()`, it is ensured that row item order will always correspond to the header item order. Before appending the root item to `m_ui->listFound`, it is passed to `FillMissingInfo()`, which completes its children with some default values (layouter and validator name) that might not be found in the input file. When all items are added into the `m_ui->listFound` table, user can select arbitrary number of them and move them to the `m_ui->listChosen` table or vice versa. Handlers of mouse double clicks and handlers of corresponding buttons (>>>, <<<) utilize `Move()` function. Since user can select arbitrary combination of multiple root items (willing to open all graphs in the file) or only some of their children, `Move()` must very carefully create, destroy and copy items in both tables. After the user is done and hits *Open* button, `on_buttonOpen_clicked()` builds `SolutionInfo` structure for every item in the `m_ui->listFound` and appends it to the `acceptedSolutions` list. `SolutionInfo` is intended as a hint for `GraphView` constructor – where to find a graph (file name, file position), how to parse it (parser name), how to interpret it (validator name) and how to lay it out (layouter name). Note that file name is not located in every table item, but only in tool tips of root items (refer to `Parser` interface). Publicly accessible `acceptedSolutions` list acts as output storage of `OpenDialog`.

There is one tiny feature with quite complicated implementation, which should be described. Since input file might not specify validators for its graphs, attempt to open larger file with some default validator might appear to be very slow because of intensive error logging. In that case, user would be also forced to change the validator manually in every opened `GraphView`. Assuming the user knows what validator should be used for a given file; there is a possibility to change it directly in `OpenDialog` before actual opening. User selects multiple (root or normal) items in the table and by right mouse button click invokes `customContextMenuRequested()`, which is a signal of `QTreeWidget` connected to `ShowContextMenu()` slot of `OpenDialog`. `ShowContextMenu()` opens `m_contextMenu` (`QMenu` instance), whose actions correspond to validator names obtained from `Factory`, and sets `m_senderTreeWidget` to remember what `QTreeWidget` the context menu was requested from. Every menu action is connected through `m_signalMapperValidators` (`QSignalMapper` instance) to `SetValidator()` slot. `SetValidator()` discovers all selected items in `m_senderTreeWidget` table and alters their validator column with given validator name.

3.6.3 Help Dialog

Reference: *helpdialog.h*, *helpdialog.cpp*, *helpdialog.ui*

`HelpDialog` is a class, which inherits `QDialog` and has its GUI part accessible through `m_ui` pointer. Class acts as a simple help viewer of the file *index.html* in the folder `/../doc/` (relatively to the executable). Since it is expected that help file is coded in HTML, class uses `QTextBrowser` (`m_ui->textBrowser`) for the rendering. `HelpDialog` currently does not support text searching and indexing. Thus, help file should contain table of contents and should be well structured by usage of hypertext links. Note that `HelpDialog` is non-modal and has no parent (when created by `GraphRec`) in order to be accessible even if other modal dialog is shown.

3.7 Persistent Settings

GraphRec uses Qt multiplatform approach to save settings persistently between user sessions. `QSettings` class provides abstraction for uniform access to settings, which are saved in various locations – registry on Windows (`HKEY_CURRENT_USER\Software\` or `HKEY_LOCAL_MACHINE\Software\` or `HKEY_LOCAL_MACHINE\Software\WOW6432node`) or conf files on Unix (`$HOME/.config/` or `/etc/xdg/`). Because application name (*GraphRec*) and domain

(*koupy.net*) is set explicitly in the `GraphRec` class constructor, it is possible to work with settings anywhere in the code without specifying it again. It is sufficient to make `QSettings` instance on the stack and then call its `value()` function for fetching entries or `setValue()` function for saving entries. GraphRec settings can be reviewed either in constructors of GUI classes or in functions that prepares/deletes modal dialogs.

4 File Formats

4.1 Multirobot

Reference: *multirobotparser.cpp*, *multirobotsaver.cpp*

Multirobot file format is the initial format supported by GraphRec. It is derived from the output of *multirobot*, the software for solving *multirobot path planning* and *pebble motion on graph*, developed by RNDr. Pavel Surynek, Ph.D. (http://ktiml.mff.cuni.cz/~surynek/). Advantage of having common format is that there is no need for any converter. There was also no time wasted on developing such converter. However, chosen approach has also some disadvantages – mainly the fact that some data fields are not relevant for GraphRec (which results in unnecessarily large files).

Format specification consists of grammar (in *Extended Backus–Naur Form* with case insensitive terminals) and its semantic description. Note that there are fields defined by grammar but irrelevant for GraphRec. When it happens that whole line is irrelevant, it is not even specified by the grammar and considered as an auxiliary non-terminal. Since auxiliary non-terminal has no exact specification, grammar should be used only as a reference due to its incompleteness. It can be assumed that line standing behind auxiliary non-terminal cannot interfere with the rest of the grammar. GraphRec puts also some optional extensions to the original format. These extensions include information about node positions, graph coloring, viewport state and validation mode. GraphRec can open either original or extended file but can save only extended file.

4.1.1 Grammar

```
file = { graph } , '<EOF>' ;
graph = { aux } , [ id ] ,
        { aux } , vertex block ,
        { aux } , edge block ,
        { aux } , [ circle block ] ,
        { aux } , [ validator block ] ,
        { aux } , [ color block ] ,
        { aux } , [ position block ] ,
        { aux } , solution block ,
        { aux } , length ,
        { aux } ;
id = 'id:' , uint , nl ;
vertex block = 'V =' , nl , { vertex }, nl ;
vertex = '(' , uint , ':' , uint , ')' ,
         '[' , sint , ':' , sint , ':' , sint '] ' ,
         { uintwh } , nl ;
edge block = 'E =' , nl , { edge }, nl ;
edge= '{' , uint , ',' , uint , '} (' , uint , ')' , nl ;
circle block = 'C =' , nl , { circle }, nl ;
circle = uintwh , '(' , uint , ',' , uint , '): ' , { uintwh } ,
         ' [' , { uintwh } , '] ' , '{' , { uintwh } , '}' , nl ;
validator block = 'MOD =' , nl , ( 'M:IMMEDIATE' | 'M:TRANSITIVE' ) , nl ;
color block = 'COL =' , nl , [ scene ] , [ borders ] ,
              [ highlight ] , { color } , nl ;
scene = 'B_SCN:A:' , color value , nl ;
borders = 'P_BRD:A:' , color value , nl ;
highlight = 'P_HLT:A:' , color value , nl ;
color = ( 'B' | 'P' ) , '_' , ( 'EMP' | 'INH' | 'FIN' ) ,
        ':' , uint , ':' , color value , nl ;
position block = 'POS =' , nl , [ matrix ] , [ angle ] ,
                 [ center ] , { position } , nl ;
matrix = ' MATRIX:' , float , ':' , float , ':' , float , ':' ,
```

```
         float , ':' , float , ':' , float , nl ;
angle = 'ANGLE:' , float , nl ;
center = 'CENTER:X' , float , ':Y' , float , nl ;
position = uint , ':X' , float , ':Y' , float , nl ;
solution block = 'Solution' , nl , { move } , nl ;
move = uintwh , '---> ' , uintwh , [ move extension ] ,
       '(' , uint , ')' , nl ;
move extension = '[' , sint , ' ---> ' , sint , '] ' ;
length = 'Length:' , uint , nl ;
aux = ? unknown additional information ? , nl ;
nl = new line , { new line } ;
new line = '<LF>' | '<CR>' | '<LF>' , '<CR>' | '<CR>' , '<LF>' ;
numeral = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
uint = numeral , { numeral } ;
uintwh = uint , ' ' ;
sint = [ '+' ] , uint | '-' , uint ;
float = sint , '.' , uint | sint ;
hnumeral = numeral | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' ;
color value = '#' , hnumeral , hnumeral , hnumeral , hnumeral ,
              hnumeral , hnumeral ;
```

4.1.2 Description

- `(<node_id>:<IGNORED>)[<initial_entity_id>:<IGNORED>:<IGNORED>]` stands for vertex definition. Note that entity identifications less or equal zero are reserved for empty nodes.
- `{<source_node_id>,<destination_node_id>} (<IGNORED>)` stands for edge definition.
- `<circle_id> (<source_node_id>,<destination_node_id>): <whole_circle> [<new_arc>] {<existing_arc>}` stands for circle definition. Last three tokens are lists of node identifications. Circle is created by joining new arc to the source and destination node, both of which are joints of existing arc.
- `M:IMMEDIATE` selects validator for *Pebble motion on graph*.
- `M:TRANSITIVE` selects validator for *Multirobot path planning*.
- `B_SCN:A:<color>` is background color for graph view scene. Letter *A* stands for *all*, but has no practical meaning in this context.
- `P_BRD:A:<color>` is color for edges and node borders. Letter *A* stands for *all*, but has no practical meaning in this context.
- `P_HLT:A:` is color for edge highlighting. Letter *A* stands for *all*, but has no practical meaning in this context.
- `B_EMP:<node_id>:<color>` is background color for empty node.
- `P_EMP:<node_id>:<color>` is foreground color for empty node.
- `B_INH:<entity_id>:<color>` is background color for entity in non-final position. *INH* stands for *inhabited*.
- `P_INH:<entity_id>:<color>` is foreground color for entity in non-final position. *INH* stands for *inhabited*.
- `B_FIN:<entity_id>:<color>` is background color for entity in final position. *FIN* stands for *final*.
- `P_FIN:<entity_id>:<color>` is foreground color for entity in final position. *FIN* stands for *final*.
- `MATRIX:<m11>:<m12>:<m21>:<m22>:<dx>:<dy>` is transformation matrix of the scene.
- `ANGLE:<angle>` is rotation of the scene (in degrees). Currently **obsolete**, because scene is no longer rotated – rotation is saved in node positions instead.
- `CENTER:X<x>:Y<y>` is point in the scene that is aligned to the center of viewport.
- `<node_id>:X<x>:Y<y>` is position of specified node.
- `<source_node_id> ---> <destination_node_id> [<IGNORED> ---> <IGNORED>] (<time_step>)` defines one particular move of unspecified entity between two specified nodes at specified time step. Direction of arrow does not necessarily match direction of move – if suggested direction is evaluated as invalid by validator, there is a chance that the inverse direction would be evaluated as valid. Note that these lines might not be sorted by time step in the input file.

4.1.3 Example

```
id:1
V =
```

```
(1:0)[1:0:0]
(2:0)[2:0:0]
(3:0)[3:0:0]
E =
{1,2} (0)
{2,3} (0)
{3,1} (0)
MOD =
M:TRANSITIVE
COL =
B_SCN:A:#ffffff
P_BRD:A:#000000
P_HLT:A:#00ffff
B_EMP:1:#ffaa00
B_EMP:2:#ffaa00
B_EMP:3:#ffaa00
P_EMP:1:#000000
P_EMP:2:#000000
P_EMP:3:#000000
B_INH:1:#0000ff
B_INH:2:#ff0000
B_INH:3:#00ff00
P_INH:1:#ffffff
P_INH:2:#ffffff
P_INH:3:#ffffff
B_FIN:1:#00007f
B_FIN:2:#aa0000
B_FIN:3:#005500
P_FIN:1:#ffffff
P_FIN:2:#ffffff
P_FIN:3:#ffffff
POS =
MATRIX:1.68179:0:0:1.68179:0:0
ANGLE:0
CENTER:X-7.13524:Y-35.6762
1:X-18.7568:Y-8.08399
2:X4.0907:Y-71.4452
3:X-62.2215:Y-59.842
Solution
2 ---> 3 [0 ---> 0] (0)
3 ---> 1 [0 ---> 0] (0)
1 ---> 2 [0 ---> 0] (0)
3 ---> 1 [0 ---> 0] (1)
2 ---> 3 [0 ---> 0] (1)
1 ---> 2 [0 ---> 0] (1)
3 ---> 1 [0 ---> 0] (2)
2 ---> 3 [0 ---> 0] (2)
1 ---> 2 [0 ---> 0] (2)
Length:9
```

4.2 GraphRec

Reference: *graphrecparser.cpp*, *graphrecsaver.cpp*

GraphRec file format is refined alternative to the *Multirobot* file format, which is burdened by compatibility redundancies. Format is specified in XML and is designed against requirement to provide better locality and encapsulation of information than *Multirobot* format.

4.2.1 Description

While reading this section, it is recommended to refer to the example below. Note that optional tags or attributes are enclosed in square brackets. File is composed from XML header, document type `<!DOCTYPE graphrec>` and single root element `<graphrec version="<version_number>"></graphrec>`. Root element acts as a container for one or more `<solution [id="<solution_id>"]></solution>` definitions, which are further composed from following elements:

- `[<scene [bg="<background_color>"]></scene>]` contains scene definition:
  - `[<viewport x="<x_position>" y="<y_position>"/>]` specifies point in the scene that is aligned to the center of viewport.
  - `[<matrix m11="<m11>" m12="<m12>" m21="<m21>" m22="<m22>" dx="<dx>" dy="<dy>"/>]` defines transformation matrix of the scene.
- `<graph></graph>` contains graph definition:
  - `<entity id="<entity_id>" [bg="<background_color>"] [bgf="<final_background_color>"] [fg="<foreground_color>"] [fgf="<final_foreground_color>"]/>` stands for entity definition. Note that entity identification must be greater than zero.
  - `<node id="<node_id>" [ent="<initial_entity_id>"] [x="<x_position>" y="<y_position>"] [bg="<background_color>"] [fg="<foreground_color>"] [bnd="<boundary_color>"]/>` stands for vertex definition. Note that entity identification must be greater than zero.
  - `<edge n1="<first_node_id>" n2="<second_node_id>" [ln="<line_color>"] [hgl="<highlight_color>"]/>` stands for edge definition.
- `<scenario [validator="<validator_name>"]></scenario>` contains all movements off the solution:
  - `<move tms="<time_step>" src="<source_node_id>" dst="<destination_node_id>"/>` defines one particular move of unspecified entity between two specified nodes at specified time step. Direction implied by node atrributes does not necessarily match direction of move – if suggested direction is evaluated as invalid by validator, there is a chance that the inverse direction would be evaluated as valid. Note that move definitions might not be sorted by time step in the input file.

4.2.2 Example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE graphrec>
<graphrec version="1.0">
    <solution id="1">
        <scene bg="#ffffff">
            <viewport x="-1.18921" y="-29.7302"/>
            <matrix m11="1.68179" m12="0" m21="0"
                    m22="1.68179" dx="0" dy="0"/>
        </scene>
        <graph>
            <entity id="1" bg="#0000ff" bgf="#00007f"
                           fg="#ffffff" fgf="#ffffff"/>
            <entity id="2" bg="#ff0000" bgf="#aa0000"
                           fg="#ffffff" fgf="#ffffff"/>
            <entity id="3" bg="#00ff00" bgf="#005500"
                           fg="#ffffff" fgf="#ffffff"/>
            <node id="1" ent="1" x="-18.7568" y="-8.08399"
                           bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <node id="2" ent="2" x="4.0907" y="-71.4452"
                           bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <node id="3" ent="3" x="-62.2215" y="-59.842"
                            bg="#ffaa00" fg="#000000" bnd="#000000"/>
            <edge n1="1" n2="2" ln="#000000" hgl="#00ffff"/>
            <edge n1="2" n2="3" ln="#000000" hgl="#00ffff"/>
            <edge n1="3" n2="1" ln="#000000" hgl="#00ffff"/>
        </graph>
        <scenario validator="Multirobot">
            <move tms="0" src="3" dst="1"/>
            <move tms="0" src="1" dst="2"/>
            <move tms="0" src="2" dst="3"/>
            <move tms="1" src="1" dst="2"/>
            <move tms="1" src="2" dst="3"/>
            <move tms="1" src="3" dst="1"/>
            <move tms="2" src="1" dst="2"/>
            <move tms="2" src="2" dst="3"/>
            <move tms="2" src="3" dst="1"/>
        </scenario>
    </solution>
```

```
</graphrec>
```