```
cd ..\plugandpaintplugins
nmake clean
qmake -config release
nmake
```

If everything compiled and linked without any errors, we will get a plugandpaint.exe executable and the pnp_basictools.dll and pnp_extrafilters.dll plugin files.

Creating the Application Package

To deploy the application, we must make sure that we copy the relevant Qt DLLs (corresponding to the Qt modules used in the application) and the Windows platform plugin, qwindows.dll, as well as the executable to the same directory tree in the release subdirectory.
In contrast to user plugins, Qt plugins have to be put into subdirectories matching the plugin type. The correct location for the platform plugin is a subdirectory named platforms. Qt Plugins section has additional information about plugins and how Qt searches for them.
Qt relies on the ICU library for unicode support. Therefore, you must include the ICU DLLs that are located in the bin directory of your Qt installation if Qt was configured to use ICU. The Qt version bundled in the Qt5 package uses ICU, so deployment is needed there. The ICU DLLs are version dependent and have to match the ones your Qt version was linked against.
If you are using ANGLE (the default) then you additionally need to include both libEGL.dll and libGLESv2.dll from Qt's 'lib' directory as well as the HLSL compiler from DirectX. The HLSL compiler library is called d3dcompiler_XX.dll where XX is the version number that ANGLE (libGLESv2) was linked against.
Remember that if your application depends on compiler specific libraries, these must be redistributed along with your application. You can check which libraries your application is linking against by using the depends tool. For more information, see the Application Dependencies section.
We'll cover the plugins shortly, but first we'll check that the application will work in a deployed environment: Either copy the executable and the Qt DLLs to a machine that doesn't have Qt or any Qt applications installed, or if you want to test on the build machine, ensure that the machine doesn't have Qt in its environment.
If the application starts without any problems, then we have successfully made a dynamically linked version of the Plug & Paint application. But the application's functionality will still be missing since we have not yet deployed the associated plugins.
Plugins work differently to normal DLLs, so we can't just copy them into the same directory as our application's executable as we did with the Qt DLLs. When looking for plugins, the application searches in a plugins subdirectory inside the directory of the application executable.
So to make the plugins available to our application, we have to create the plugins subdirectory and copy over the relevant DLLs:

```
plugins\pnp_basictools.dll
plugins\pnp_extrafilters.dll
```

An archive distributing all the Qt DLLs and application specific plugins required to run the Plug & Paint application, would have to include the following files:

Component	File Name
The executable	plugandpaint.exe
The Basic Tools plugin	plugins\pnp_basictools.dll
The ExtraFilters plugin	plugins\pnp_extrafilters.dll
The Qt Windows platform plugin	platforms\qwindows.dll
The Qt Core module	Qt5Core.dll
The Qt GUI module	Qt5Gui.dll
The Qt Widgets module	Qt5Widgets.dll

In addition, the archive must contain the following compiler specific libraries depending on your version of Visual Studio:

	VC++ 8.0 (2005)	VC++ 9.0 (2008)	VC++ 10.0 (2010)
The C run-time	msvcr80.dll	msvcr90.dll	msvcr100.dll
The C++ run-time	msvcp80.dll	msvcp90.dll	msvcp100.dll

If ICU was used, the archive must contain:

File Name		
icudtXX.dll	icuinXX.dll	icuucXX.dll

Finally, if ANGLE was used, then the archive must additionally contain:

File Name		
libEGL.dll	libGLESv2.dll	d3dcompiler_XX.dll

To verify that the application now can be successfully deployed, you can extract this archive on a machine without Qt and without any compiler installed, and try to run it.
An alternative to putting the plugins in the plugins subdirectory is to add a custom search path when you start your application using QApplication::addLibraryPath() or QApplication::setLibraryPaths().

```
qApp->addLibraryPath("C:\some\other\path");
```

One benefit of using plugins is that they can easily be made available to a whole family of applications.
It's often most convenient to add the path in the application's main() function, right after the QApplication object is created. Once the path is added, the application will search it for plugins, in addition to looking in the plugins subdirectory in the application's own directory. Any number of additional paths can be added.

Manifest files

When deploying an application compiled with Visual Studio 2005 onwards, there are some additional steps to be taken.
First, we need to copy the manifest file created when linking the application. This manifest file contains information about the application's dependencies on side-by-side assemblies, such as the runtime libraries.
The manifest file needs to be copied into the **same** folder as the application executable. You do not need to copy the manifest files for shared libraries (DLLs), since they are not used.
If the shared library has dependencies that are different from the application using it, the manifest file needs to be embedded into the DLL binary. Since Qt 4.1.3, the following CONFIG options are available for embedding manifests:

```
embed_manifest_dll
embed_manifest_exe
```

To use the options, add

```
CONFIG += embed_manifest_exe
```

to your .pro file. The embed_manifest_dll option is enabled by default. The embed_manifest_exe option is NOT enabled by default.
You can find more information about manifest files and side-by-side assemblies at the MSDN website.
The correct way to include the runtime libraries with your application is to ensure that they are installed on the end-user's system.
To install the runtime libraries on the end-user's system, you need to include the appropriate Visual C++ Redistributable Package (VCRedist) executable with your application and ensure that it is executed when the user installs your application.
For example, on an 32-bit x86-based system, you would include the vcredist_x86.exe executable. The vcredist_IA64.exe and vcredist_x64.exe executables provide the appropriate libraries for the IA64 and 64-bit x86 architectures, respectively.
Note: The application you ship must be compiled with exactly the same compiler version against the same C runtime version. This prevents deploying errors caused by different versions of the C runtime libraries.

Manual installations with Visual Studio 2008 and 2010

As well as the above details for VS 2005 and onwards, Visual Studio 2008/2010 applications may have problems when deploying manually, say to a USB stick.
The recommended procedure is to configure Qt with the -plugin-manifests option using the 'configure' tool. Then follow the guidelines for manually deploying private assemblies.
In brief the steps are

create a folder structure on the development computer that will match the target USB stick directory structure, for example '\app' and for your dlls, '\app\lib'.
on the development computer, from the appropriate 'redist' folder copy over Microsoft.VC80.CRT and Microsoft.VC80.MFC to the directories '\app' and '\app\lib' on the development PC.
xcopy the \app folder to the target USB stick.

Your application should now run. Also be aware that even with a service pack installed the Windows DLLs that are linked to will be the defaults. See the information on how to select the appropriate target DLLs.

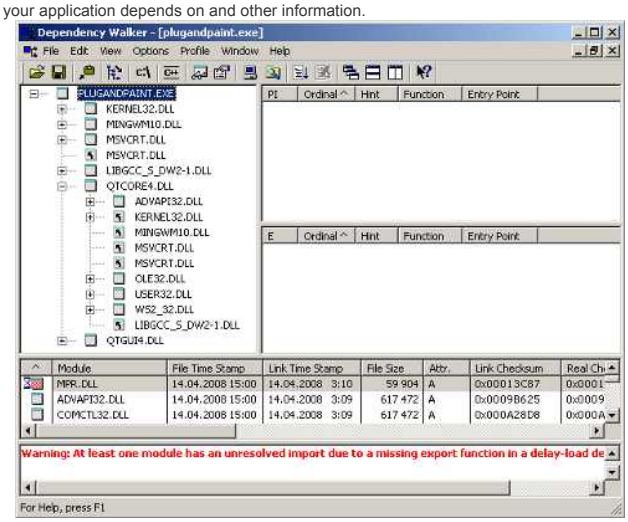
Application Dependencies

Additional Libraries

Depending on configuration, compiler specific libraries must be redistributed along with your application.
For example, if Qt is built using ANGLE, its shared libraries and the required shared libraries of the Direct X SDK need to be shipped as well.
You can check which libraries your application is linking against by using the Dependency Walker tool. All you need to do is to run it like this:

```
depends <application executable>
```

This will provide a list of the libraries that your application depends on and other information.



When looking at the release build of the Plug & Paint executable (plugandpaint.exe) with the depends tool, the tool lists the following immediate dependencies to non-system libraries:

Qt	VC++ 8.0 (2005)	VC++ 9.0 (2008)	VC++ 10.0 (2010)	MinGW
QT5CORE.DLL - The QtCore runtime	MSVCR80.DLL - The C runtime	MSVCR90.DLL - The C runtime	MSVCR100.DLL - The C runtime	MINGWM10.DLL - The MinGW run-time
QT5GUI.DLL - The QtGui runtime	MSVCP80.DLL - The C++ runtime	MSVCP90.DLL - The C++ runtime	MSVCP100.DLL - The C++ runtime	LIBGCC_S_DW2-1.DLL
QT5WIDGETS.DLL - The QtWidgets runtime				LIBSTDC++-6.dll

When looking at the plugin DLLs the exact same dependencies are listed.

Qt Plugins

All Qt GUI applications require a plugin that implements the Qt Platform Abstraction (QPA) layer in Qt 5. For Windows, the name of the platform plugin is qwindows.dll. This file must be located within a specific subdirectory (by default, platforms) under your distribution directory. Alternatively, it is possible to adjust the search path Qt uses to find its plugins, as described below.
Your application may also depend on one or more Qt plugins, such as the print support plugin, the JPEG image format plugin or a SQL driver plugin. Be sure to distribute any Qt plugins that you need with your application. Similar to the platform plugin, each type of plugin must be located within a specific subdirectory (such as printsupport, imageformats or sqldrivers) within your distribution directory.
Note: If you are deploying an application that uses Qt WebKit to display HTML pages from the World Wide Web, you should include all text codec plugins to support as many HTML encodings possible.
The search path for Qt plugins is hard-coded into the QtCore library. By default, the plugins subdirectory of the Qt installation is the first plugin search path. However, pre-determined paths like the default one have certain disadvantages. For example, they may not exist on the target machine. For that reason, you need to examine various alternatives to make sure that the Qt plugins are found:

Using qt.conf. This approach is the recommended if you have executables in different places sharing the same plugins.
Using QApplication::addLibraryPath() or QApplication::setLibraryPaths(). This approach is recommended if you only have one executable that will use the plugin.
Using a third party installation utility to change the hard-coded paths in the QtCore library.

If you add a custom path using QApplication::addLibraryPath it could look like this:

```
qApp->addLibraryPath("C:/customPath/plugins");
```

Then qApp->libraryPaths() would return something like this:

C:/customPath/plugins
C:/Qt/5.3.1/plugins
E:/myApplication/directory

The executable will look for the plugins in these directories and the same order as the QStringList returned by qApp->libraryPaths(). The newly added path is prepended to the qApp->libraryPaths() which means that it will be searched through first. However, if you use qApp->setLibraryPaths(), you will be able to determine which paths and in which order they will be searched.
The How to Create Qt Plugins document outlines the issues you need to pay attention to when building and deploying plugins for Qt applications.

The Windows Deployment Tool

The Windows deployment tool can be found in QTDIR/bin/windeployqt. It is designed to automate the process of creating a deployable folder that contains all libraries, QML imports, plugins, translations that are required to run the application from that folder. This is used to create the sandbox for Windows Runtime or an installation tree for Windows desktop applications that can be easily bundled by an installer.

```
Usage: windeployqt [options] [file]
Qt Deploy Tool 5.3.0

The simplest way to use windeployqt is to add the bin directory of your Qt
installation (e.g. <QT_DIR\bin>) to the PATH variable and then run:
  windeployqt <path-to-app-binary>
If ICU, ANGLE, etc. are not in the bin directory, they need to be in the PATH
variable. If your application uses Qt Quick, run:
  windeployqt --qmldir <path-to-app-qml-files> <path-to-app-binary>

Options:
  -?, -h, --help            Displays this help.
  -v, --version             Displays version information.
  --dir <directory>         Use directory instead of binary directory.
  --libdir <path>           Copy libraries to path.
  --debug                   Assume debug binaries.
  --release                 Assume release binaries.
  --force                   Force updating files.
  --dry-run                 Simulation mode. Behave normally, but do not
                            copy/update any files.
  --no-plugins              Skip plugin deployment.
  --no-libraries            Skip library deployment.
  --qmldir <directory>      Scan for QML-imports starting from directory.
  --no-quick-import         Skip deployment of Qt Quick imports.
  --no-translations         Skip deployment of translations.
  --no-system-d3d-compiler  Skip deployment of the system D3D compiler.
  --compiler-runtime        Deploy compiler runtime (Desktop only).
  --no-compiler-runtime     Do not deploy compiler runtime (Desktop only).
  --webkit2                 Deployment of WebKit2 (web process).
  --no-webkit2              Skip deployment of WebKit2.
  --json                    Print to stdout in JSON format.
  --list <option>           Print only the names of the files copied.
                            Available options:
                             source:   absolute path of the source files
                             target:   absolute path of the target files
                             relative: paths of the target files, relative
                                       to the target directory
                             mapping:  outputs the source and the relative
                                       target, suitable for use within an
                                       Appx mapping file
  --verbose <level>         Verbose level.

Qt libraries can be added by passing their name (-xml) or removed by passing
the name prepended by --no- (--no-xml). Available libraries:
bluetooth clucene concurrent core declarative designercomponents designer gui
clucene qthelp multimedia multimediawidgets multimediaquick network nfc opengl
positioning printsupport qml quick quickparticles script scripttools sensors
serialport sql svg test widgets winextras xml xmlpatterns

Arguments:
  [file]                    Binary or directory containing the binary.
```


Hide Notes

Notes provided by the Qt Community

Sorted by:
Rating
Date

Missing Opengl ES dlls may cause windows plugin issues

Even if you have qwindows.dll in the platforms folder, you may still see an error saying that there is no platform plugin. this could be due to missing opengl es dlls:

libEGL.dll
libGLESv2.dll

the error message is not very intuitive.

Informative
5

Votes: 1
Coverage: Qt 5

billconan
Ant Farmer
6 notes