Großer Beleg

Portierung von Qt auf Genode

Christian Prochaska

11. Februar 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Carsten Weinhold
Co-Betreuer: Dr.-Ing. Norman Feske
Dipl.-Inf. Christian Helmuth

Abstract

Genode ist eine neue Betriebssystemarchitektur, die durch die Auferlegung einer streng hierarchischen Prozessstruktur und den Einsatz von Capabilities aufbauend auf einem Mikrokern ein besonders hohes Maß an Sicherheit gewährleisten soll. Das „Genode Operating System Framework“ ist die Referenzimplementierung dieser Betriebssystemarchitektur mit Unterstützung für die Fiasco- und L4ka::Pistachio-Mikrokerne sowie Linux für Entwicklungs- und Debugging-Zwecke.

Ziel dieser Belegarbeit war es, durch die Portierung des GUI-Toolkits Qt die Grundlage für die Entwicklung einer grafischen Benutzeroberfläche für Genode auf dem Fiasco-Mikrokern zu schaffen. Es werden die einzelnen Schritte des Portierungsvorganges dargestellt - von der Integration des Qt-Sourcecodes in das Genode-Buildsystem über die Entwicklung einer kleinen C/C++-Supportbibliothek, die Anpassung der Qt-Klassen für Timer-, Synchronisations- und Thread-Funktionalität bis hin zur Entwicklung der benötigten Qt-Treiberklassen für die grafische Ausgabe und die Tastatur- und Mauseingabe über den Nitpicker GUI-Server. In der abschließenden Auswertung werden Praxistauglichkeit, Zukunftssicherheit, Optimierungsmöglichkeiten und Limitierungen der entstandenen Portierungslösung an Hand einer Beispielanwendung betrachtet sowie kurz auf die während der Portierungsarbeit gewonnenen Erfahrungen mit dem Genode-Framework eingegangen.

Inhaltsverzeichnis

1 Einleitung

Die Betriebssystemgruppe der TU-Dresden beschäftigt sich mit der Erforschung sicherer Betriebssysteme aufbauend auf einem Mikrokern. Im Rahmen dieser Forschungstätigkeiten sind im Jahre 2006 die „Genode“-Architektur und deren Referenzimplementierung „Genode Operating System Framework“ entstanden, welche vor allem durch eine streng hierarchische Prozessstruktur und den Einsatz von Capabilities ein besonders hohes Maß an Sicherheit gewährleisten sollen.

Mittlerweile ist die Entwicklung des Genode-Basissystems so weit fortgeschritten, dass ein Einsatz des Frameworks für praktische, insbesondere auch grafische Anwendungen möglich und wünschenswert ist. Für die Entwicklung grafischer Anwendungen greift man heutzutage üblicherweise auf eine Klassenbibliothek zurück, die die sichtbaren Bestandteile der Anwendung auf dem Bildschirm (also üblicherweise mindestens ein Fenster und alle darin enthaltenen Widgets[1]) in Form von Klassen abstrahiert. Diese Klassenbibliothek kann fester Bestandteil des Betriebssystems sein, wie z.B. unter Windows, oder aber auch weitestgehend plattformunabhängig aufgebaut sein wie z.B. die GTK+- oder Qt-Bibliotheken unter Linux. Für Genode existieren mit DoPE und „Scout Widgets“ bereits zwei kleine Bibliotheken, die die Entwicklung einfacher grafischer Anwendungen ermöglichen, aber im Vergleich zu den umfangreichen Klassenbibliotheken für die verbreiteten Betriebssysteme sind die Möglichkeiten dieser zwei kleinen Bibliotheken doch sehr eingeschränkt. Da sich deren Programmierschnittstellen zudem noch von den Schnittstellen der anderen Klassenbibliotheken unterscheiden, wäre die Erstellung eines neuen, sicheren Betriebssystems mit all den gewohnten grafischen Anwendungen wie Web-Browser, Mediaplayer oder Textverarbeitung auf dieser Basis äußerst arbeits- und zeitaufwändig oder müsste auf Virtualisierungsschichten in der Art von „Linux auf Genode“ zurückgreifen. Letzteres würde allerdings dem Ziel entgegenlaufen, die Trusted Computing Base einer jeden Anwendung möglichst klein zu halten.

Hier stellte sich also die Frage, ob man nicht vielleicht auch eine der bereits existierenden und weit verbreiteten Klassenbibliotheken auf Genode portieren könnte. Dadurch würde sich der Entwicklungsaufwand für die benötigten Anwendungen, die auf den anderen Betriebssystemen inzwischen schon sehr umfangreich und ausgereift sind, reduzieren. Außerdem könnte man gegebenenfalls von bereits vorhandenen Entwicklungstools für diese Bibliotheken profitieren (z.B. GUI-Editoren). Die Fülle an verfügbarer Dokumentation zu den verbreiteten Klassenbibliotheken ist ebenfalls nicht zu unterschätzen.

Als vielversprechendster Kandidat für eine mögliche Portierung stellte sich schnell die Klassenbibliothek Qt der norwegischen Firma Trolltech heraus. Diese Bibliothek zeichnet sich durch eine Vielzahl von Widgets und nützliche Entwicklungstools aus, vor allem aber auch durch ihre Plattformunabhängigkeit und mit der Variante „Qt for Embedded Linux“ durch eine besonders geringe Abhängigkeit vom unterliegenden Betriebssystem, um den Einsatz auch auf eingebetteten Systemen zu ermöglichen, die in ihrer angebotenen Funktionalität oftmals doch recht eingeschränkt sind. Nicht zuletzt versprachen die positiven Ergebnisse einer bereits im Jahr 2004 erfolgten Portierung von Qt auf das ebenfalls an der TU-Dresden entwickelte Betriebssystem DROPS Erfolg, Qt im Rahmen eines Großen Beleges auch auf Genode portieren zu können und dadurch eine komfortable Entwicklung moderner grafischer Anwendungen sowie die einfache Portierung der Vielzahl bereits vorhandener Qt-Anwendungen für diese neue Betriebssystemplattform zu ermöglichen.

Im folgenden Kapitel werden das „Genode OS Framework“ und Qt genauer betrachtet. Anschließend werden in Kapitel 3 einige alternative GUI-Toolkits vorgestellt. Kapitel 4 befasst sich mit den genauen Details der Portierung und in Kapitel 5 findet eine Auswertung der Arbeit statt.

1 grafische Steuerelemente, die Informationen darstellen und Benutzereingaben entgegennehmen können

2 Grundlagen

2.1 Das Genode OS Framework

Genode[1][2] ist eine neue, an der TU Dresden entwickelte Betriebssystemarchitektur, die ein sicheres Zusammenspiel von Betriebssystemdiensten (z.B. Gerätetreibern und Dateisystemen), Anwendungen mit besonderen Sicherheitsanforderungen sowie sonstigen Anwendungen auf einem Mikrokern gewährleisten soll. Dieses Ziel konnte bisher noch nicht vollständig erreicht werden, da ein Mikrokern zwar bereits eine sichere Isolation von Komponenten erlaubt, er aber durch das Prinzip der Trennung von Mechanismus und Policy selbst keine Sicherheitspolicy implementiert und eine Implementierung der Policy in einer zentralen Sicherheitskomponente mit zunehmender Anzahl von zu kontrollierenden Objekten im System immer komplexer und damit fehleranfälliger wird. Dieses Komplexitätsproblem soll mit Genode durch die Umsetzung einer baumförmigen Prozessstruktur gelöst werden, welche eine Verteilung der Sicherheitspolicy auf mehrere Hierarchie-Ebenen und unabhängige Teilbäume ermöglicht, ähnlich wie es bei großen Unternehmen der Fall ist. Zusätzlich wird die Kommunikation zwischen Prozessen und der Zugriff auf Objekte, die von anderen Prozessen zur Verfügung gestellt werden, durch Capabilities geschützt.[3]

Um ein höchstmögliches Maß an Sicherheit zu gewährleisten, darf ein Prozess in diesem System nach seiner Erzeugung vorerst nur mit seinem Elternprozess über eine minimale, standardisierte Schnittstelle kommunizieren. Diese Schnittstelle bietet dem Kindprozess die Möglichkeiten an:

- eine Kommunikationserlaubnis (Capability) zu einem von einem anderen Prozess angebotenen Dienst anzufordern bzw. wieder zurückzugeben
- selbst einen Dienst anzumelden, der daraufhin von anderen Prozessen genutzt werden kann
- einen Teil der eigenen Ressourcen (z.B. Arbeitsspeicher) an einen anderen Prozess, zu dem eine Kommunikationsbeziehung besteht, zu transferieren
- sich zu beenden

Der Elternprozess kann dabei jeweils entsprechend seinen eigenen Sicherheitsrichtlinien entscheiden, ob und wie er die gewünschte Aktion durchführt. Er könnte z.B. nur für von ihm selbst oder von einem seiner Kindprozesse angebotene Dienste eine Kommunikationserlaubnis gewähren und Kontaktaufnahmen zu Diensten außerhalb seiner Einflusssphäre unterbinden. Des Weiteren hat er die Möglichkeit, die Liste der Parameter, die der Kindprozess dem gewünschten Dienst bei der Anforderung der Kommunikationserlaubnis übermittelt, zu analysieren und gegebenenfalls zu verändern.

Zur Verhinderung von Denial-of-Service durch übermäßigen Ressourcenverbrauch muss zusätzlich sichergestellt werden, dass ein einzelner Prozess nicht beliebig viele Ressourcen allokieren kann. Dies gilt nicht nur für Ressourcen, die durch den Prozess selbst direkt allokiert werden, sondern auch für solche, die indirekt von einem anderen Prozess für die Erbringung einer Dienstleistung im Auftrag dieses Prozesses aufgebracht werden müssen. Diese Vorgabe wird in der Genode-Architektur erfüllt, indem jeder Prozess nur so viele Ressourcen beanspruchen kann, wie er von seinem Elternprozess oder einem anderen Prozess aus dessen eigenem Ressourcenvorrat transferiert bekommen hat. Ressourcen, die ein Dienst für die Erbringung seiner Dienstleistung für einen anderen Prozess zusätzlich benötigt, müssen vorher vom Dienstnutzer an den Dienst transferiert werden. Nach Erbringung der Dienstleistung werden diese Ressourcen wieder an den Dienstnutzer zurückgegeben.

Das „Genode OS Framework“ ist die Referenzimplementierung der beschriebenen Genode-Architektur. Es wurde in C++ programmiert und kann sowohl mit dem Fiasco-Mikrokern auf der x86- und ARM-Architektur eingesetzt werden als auch unter Linux. Seit kurzem wird auch der L4ka::Pistachio-Kern unterstützt. Das Framework stellt Basisdienste für die Erzeugung von Prozessen und Threads, für die Allokation von Speicherbereichen, für die Verwaltung von Adressräumen, für den Lesezugriff auf zur Bootzeit übergebene Dateien, für die Nutzung von Memory Mapped I/O und für die Ausgabe von Debugmeldungen bereit. Darüber hinaus existieren unter Anderem ein Timer-Dienst sowie der GUI-Server Nitpicker[4] zusammen mit den von ihm benötigten Diensten für Tastatur- und Mauseingabe und einem Framebuffer-Dienst für die grafische Ausgabe. Für die komfortable Nutzung dieser Dienste in eigenen Anwendungen stehen jeweils entsprechende C++-Klassen in statisch linkbaren Bibliotheken zur Verfügung. Seit Mai 2008 wird das Framework von der Firma „Genode Labs“[5], einer Ausgründung der TU-Dresden, weiterentwickelt und soll zukünftig auch als Produkt vertrieben werden.

2.2 Qt

2.2.1 Qt im Überblick

Qt ist ein von der norwegischen Firma Trolltech entwickeltes Framework für die Erstellung plattformübergreifender grafischer Anwendungen. Es umfasst eine umfangreiche C++-Klassenbibliothek sowie verschiedene Werkzeuge, welche die Anwendungsentwicklung vereinfachen sollen. „Plattformübergreifend“ bedeutet, dass eine Qt-Anwendung ohne Änderungen am Sourcecode auf allen Plattformen lauffähig ist, auf denen Qt verfügbar ist. Zu den von Qt unterstützten Plattformen gehören Linux/X11, Windows, Mac OS X sowie Embedded Linux. Seit der Version 4.4, die während der Bearbeitung dieses Beleges von Trolltech veröffentlicht wurde, werden auch Windows CE und Windows Mobile unterstützt. Die Plattformunabhängigkeit von Qt-Anwendungen kann natürlich nur dann gewährleistet werden, wenn keine plattformspezifischen Betriebssystemfunktionen aufgerufen werden. Aus diesem Grund bietet Qt Abstraktionen für die wichtigsten Betriebssystemfunktionen in Form von Klassen an.

Die Klassenbibliothek ist modular aus mehreren Einzelbibliotheken aufgebaut, die statisch oder dynamisch zur Anwendung gelinkt werden können und jeweils ein bestimmtes Themengebiet abdecken:

- **QtCore** enthält die Qt-Kernkomponenten, mit denen bereits nicht-grafische Qt-Anwendungen entwickelt werden können. Dazu gehören neben betriebssystemunabhängigen Komponenten wie Container- und Stringklassen auch Klassen für die Abstraktion von Betriebssystemelementen wie Prozesse, Threads, Synchronisationsmechanismen, Timer, Shared Memory, Dateien und Verzeichnisse. Alle anderen Module hängen vom QtCore-Modul ab.
- **QtGui** stellt GUI-Funktionalität bereit. Das Modul bietet eine Vielzahl von Widgets wie Fenster, Buttons, Scrollleisten, Texteingabefelder usw. für die Erstellung mächtiger und einfach zu bedienender grafischer Benutzeroberflächen sowie Unterstützung für die Anzeige vieler gängiger Bildformate wie PNG oder JPEG.
- **QtNetwork** enthält Klassen für die Netzwerkprogrammierung. Es werden verschiedene Netzwerkprotokolle wie IPv4, IPv6, UDP, TCP, HTTP und FTP unterstützt.
- **QtWebKit** enthält eine Web Browser-Engine, mit welcher (X)HTML-Dokumente in Qt-Anwendungen dargestellt und auch bearbeitet werden können.
- **QtXML** enthält Klassen für die Verarbeitung von XML-Dokumenten.
- **QtSvg** ermöglicht die Generierung und Darstellung von Vektorgrafiken im SVG-Format.

- **Phonon** ermöglicht die Wiedergabe von Multimedia-Inhalten.
- **QtSQL** enthält Klassen für die Anbindung an Datenbanken.
- **QtOpenGL** ermöglicht die Verwendung des OpenGL 3D-APIs in Qt-Anwendungen.
- **QtUiTools** ermöglicht das Laden von Benutzerschnittstellenbeschreibungen, die mit der Qt-Designer-Anwendung (siehe Seite 8) erstellt wurden, zur Laufzeit.
- **QtScript** enthält eine ECMAScript-kompatible Scripting-Engine, die eine flexible Erweiterung der Funktionalität von Qt-Anwendungen ermöglichen kann.

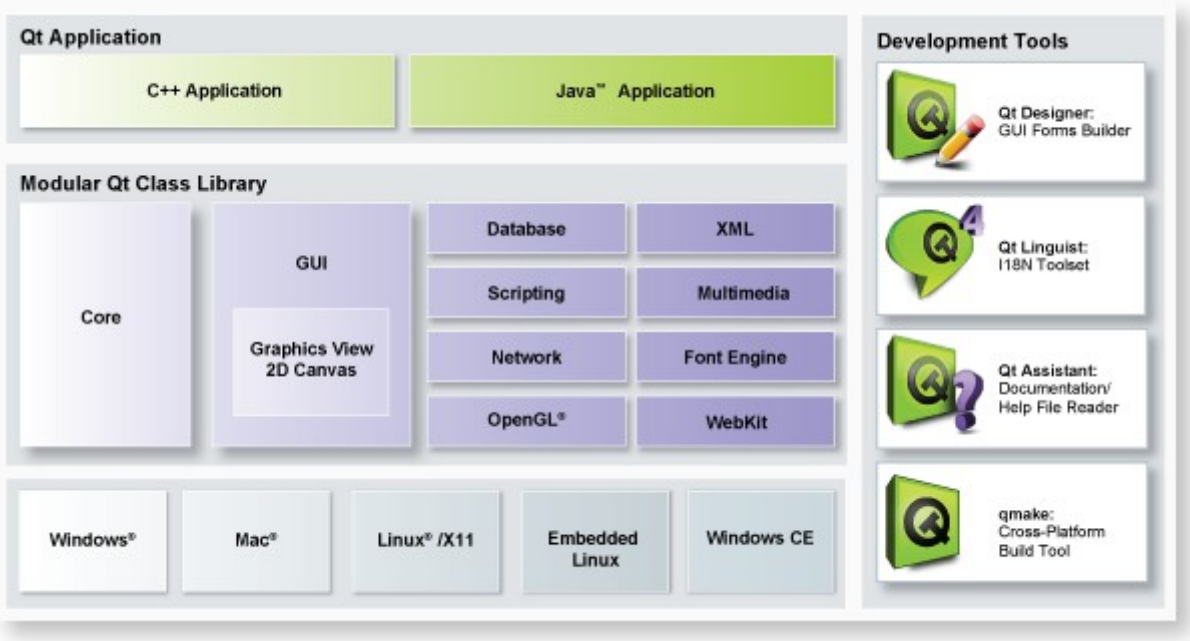

Abbildung 1: Qt-Architektur (Quelle: Trolltech) – Die einzelnen Komponenten werden im Text genauer erläutert.

Neben der Klassenbibliothek spielen auch verschiedene Werkzeuge eine bedeutende Rolle:

Das wichtigste Werkzeug ist der **Meta-Object Compiler (moc**[6]**)**. Er stellt eine Art Präprozessor dar, der dafür zuständig ist, spezielle Qt-Spracherweiterungen in gültigen C++-Code zu transformieren. Zu diesen Spracherweiterungen gehört z.B. der *Signals und Slots*-Mechanismus, der eine lose und typsichere Kopplung von Objekten (im Gegensatz zur Verwendung typunsicherer Callbacks) ermöglicht. Durch *Properties* können Objekte zur Laufzeit um zusätzliche Attribute erweitert werden und sie ermöglichen es, verschiedene Widget-Eigenschaften in der Qt Designer-Anwendung festzulegen. Und schließlich bietet sich durch das Meta-Object-System auch die die Möglichkeit, *Typinformationen* und *Vererbungsbeziehungen* zwischen Objekten zur Laufzeit zu ermitteln sowie Objekte zur Laufzeit in kompatible Typen zu konvertieren (*Dynamic Casting*), ohne die Unterstützung von RTTI (Run-Time Type Information) durch den C++-Compiler zu benötigen. Voraussetzung für die Nutzung dieser Spracherweiterungen ist, dass die jeweilige Klasse von der Klasse `QObject` erbt und in der Klassendeklaration neben den neuen Schlüsselwörtern das Makro `Q_OBJECT` enthält. Der Meta-Object Compiler generiert dann aus dieser erweiterten Klassendeklaration ein sogenanntes Meta-Objekt, welches die zusätzliche, klassenspezifische Meta-Information und -Funktionalität auf C++-Primitive abbildet und zur Programmlaufzeit bereitstellt.

Der **Resource Compiler** (**rcc**[7]) ermöglicht es, binäre Dateien in die ausführbare Datei der Anwendung einzubetten. Der Zugriff auf diese Dateien kann dann innerhalb der Anwendung wie auf normale Dateien des Dateisystem erfolgen, wobei der Pfad zu einer solchen Ressourcendatei zur Unterscheidung immer mit einem Doppelpunkt beginnt. In der Genode-Portierung wird von dieser Möglichkeit beispielsweise Gebrauch gemacht, um einen Font in grafische Anwendungen einzubetten, da Genode noch kein reguläres Dateisystem unterstützt, aus welchem der Font sonst geladen werden könnte.

Der **User Interface Compiler** (**uic**[8]) generiert aus den vom Qt Designer erzeugten Benutzerschnittstellenbeschreibungen im XML-Format C++-Header-Dateien, die kompiliert und in die Anwendung eingebunden werden können. Alternativ können die XML-Schnittstellenbeschreibungen auch mit Hilfe der QtUiTools-Bibliothek zur Programmlaufzeit geladen werden.

Da auf den verschiedenen von Qt unterstützten Plattformen unterschiedliche Compiler und Buildwerkzeuge üblich sind (z.B. `gcc` und GNU `make` unter Linux und Visual C++ und `nmake` unter Windows), ist neben einem einheitlichen Programmcode auch eine portable Beschreibung des Buildvorganges wünschenswert. Qt bietet eine solche Beschreibungsmöglichkeit in Form deklarativer Projektdateien an. Aus einer oder mehreren dieser einheitlichen Projektdateien generiert das **qmake**[9]-Programm dann während des Buildvorganges auf einer bestimmten Plattform automatisch die entsprechenden plattformspezifischen Makefiles.

Der **Qt Designer**[10] (siehe Abb. 2) ermöglicht die komfortable grafische Erstellung von Benutzerschnittstellen für Qt-Anwendungen. Ausgehend von einem leeren Dialogfenster kann der Benutzer die gewünschten Widgets aus einer Liste auswählen und mit der Maus in das Dialogfenster platzieren. In separaten Fensterbereichen können anschließend Widget-Eigenschaften angepasst, Layouts eingerichtet, Signals und Slots der platzierten Widgets miteinander verbunden und Ressourcendateien erstellt werden. Die Beschreibung der erstellten Benutzeroberfläche wird dann abschließend in einer XML-Datei gespeichert und kann wie bereits erwähnt entweder zur Compilezeit mit Hilfe des User Interface Compilers oder zur Programmlaufzeit mit Hilfe der QtUiTools-Bibliothek in die Anwendung eingebunden werden.

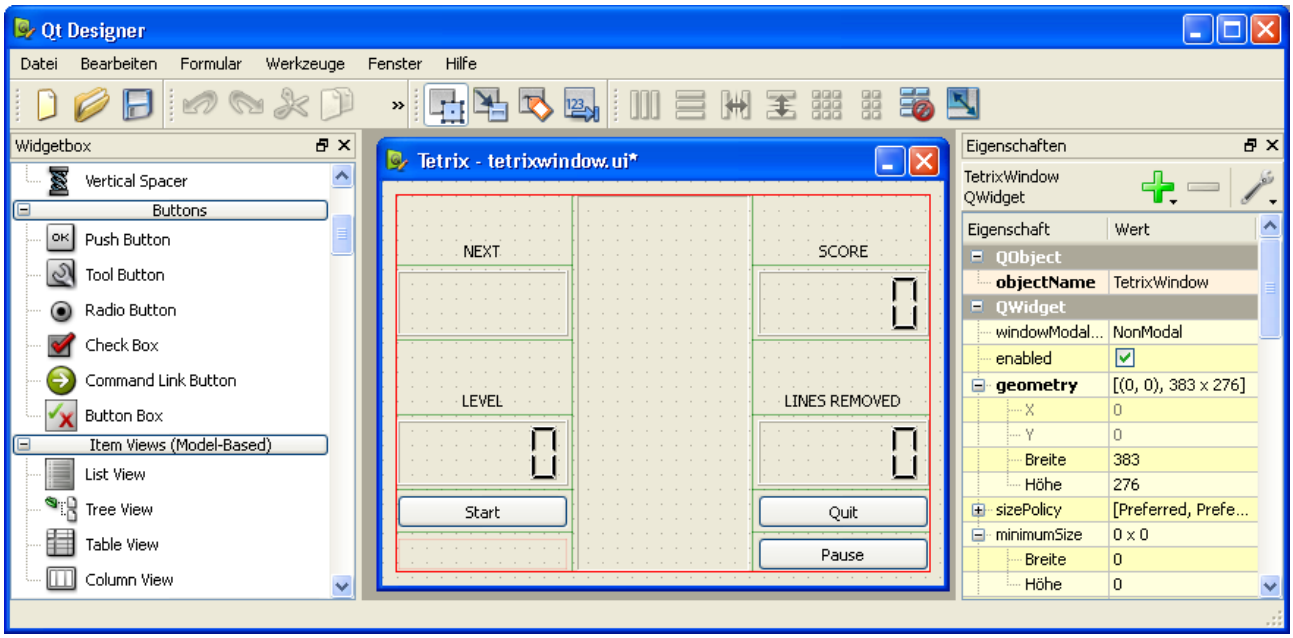

Abbildung 2: Qt Designer

Mit dem **Qt Linguist**[11] (siehe Abb. 3) können Qt-Anwendungen auf komfortable Weise in verschiedene Sprachen übersetzt werden.

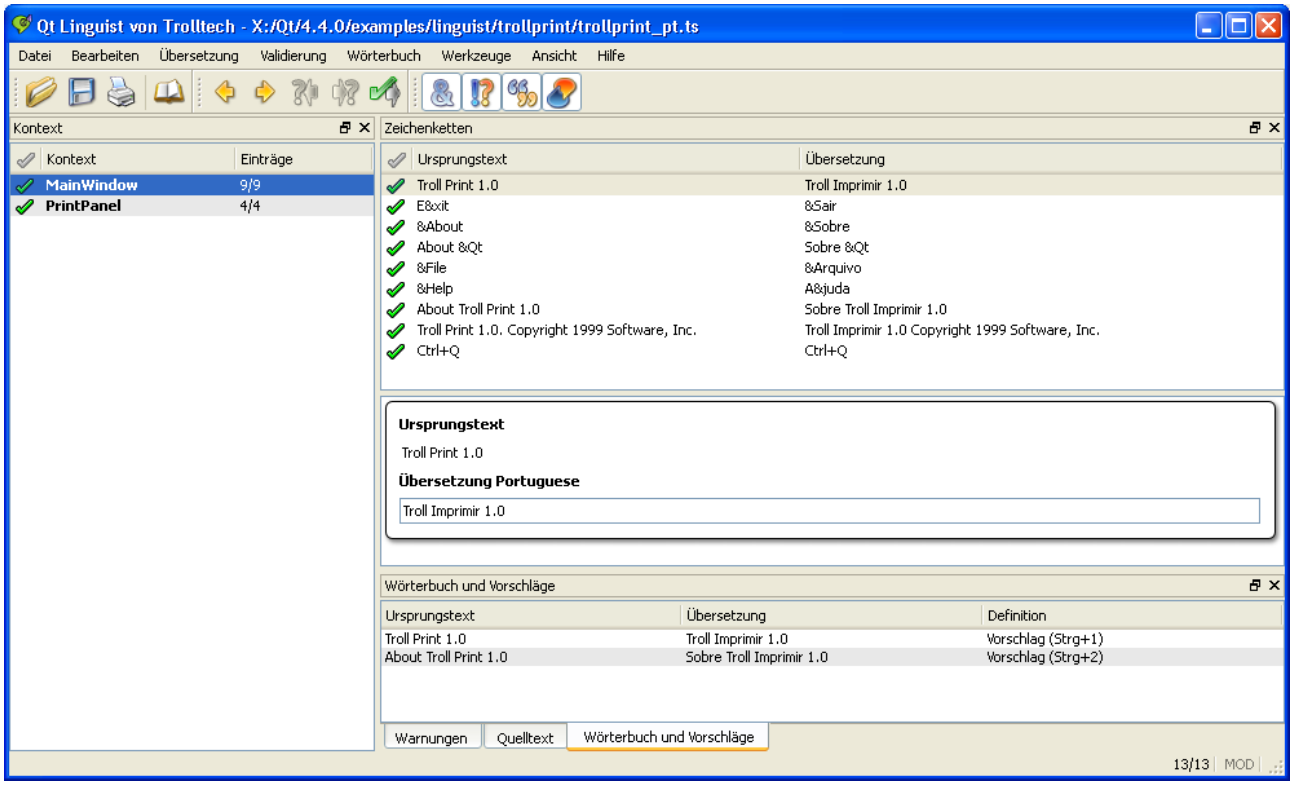

Abbildung 3: Qt Linguist

Der **Qt Assistant**[12] (siehe Abb. 4) schließlich ist ein Browser für Onlinedokumentationen. Er wird für die Dokumentation von Qt eingesetzt und kann auch zur Dokumentation von eigenen Anwendungen verwendet werden.

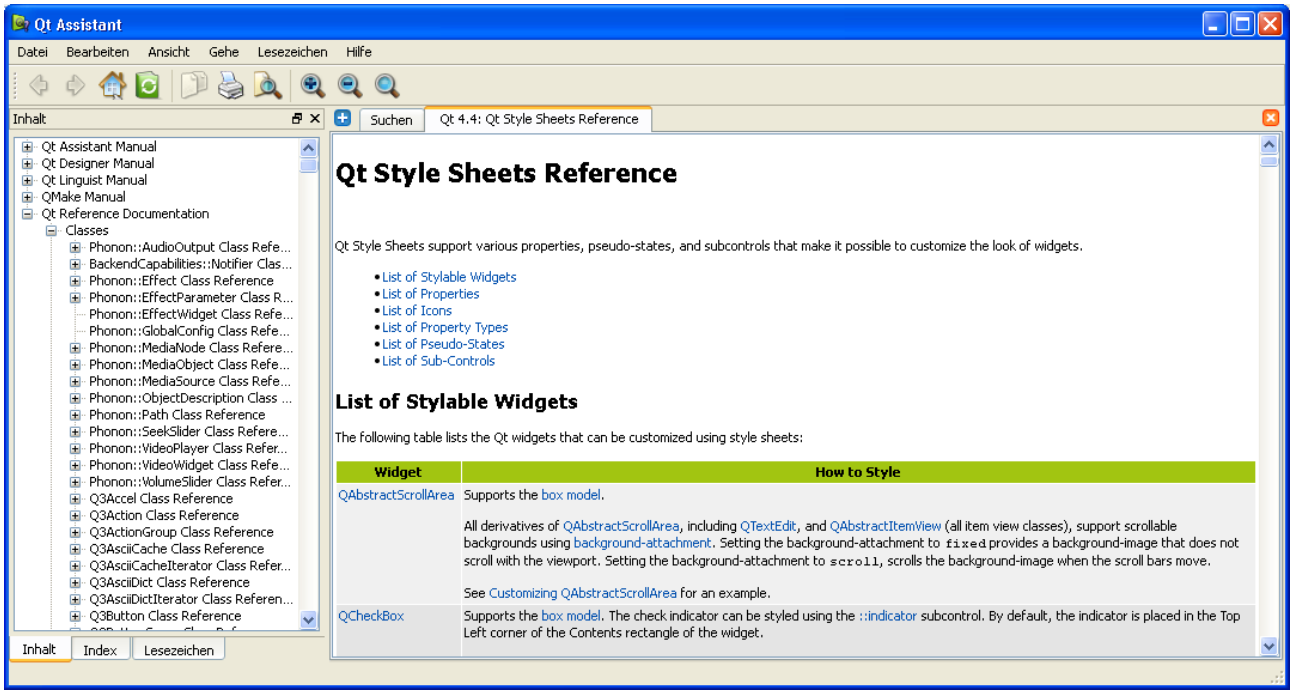

Abbildung 4: Qt Assistant

2.2.2 Qt-Variante „Qt for Embedded Linux“

Die Qt-Variante „Qt for Embedded Linux“[13] zeichnet sich gegenüber den Qt-Versionen für Linux/X11, Windows oder Mac OS X dadurch aus, dass sie für die grafische Darstellung und Benutzereingabe nicht auf ein bereits vorhandenes Fenstersystem wie z.B. X11 zurückgreift, sondern stattdessen ein eigenes leichtgewichtiges Fenstersystem, das „Qt Windowing System“ (QWS), einsetzt, welches direkt auf den Linux-Frambuffer zugreift. Dadurch kann der Ressourcenverbrauch für eine grafische Benutzeroberfläche im Vergleich zum Einsatz von z.B. X11 sehr gering gehalten werden, was den Einsatz von Qt auch auf mobilen Geräten wie z.B. PDAs ermöglicht.

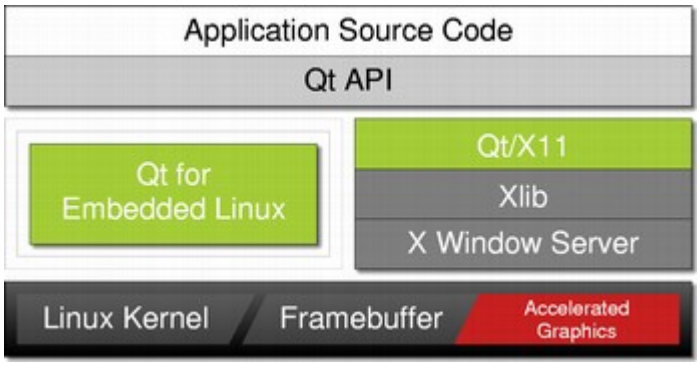

Abbildung 5: Gegenüberstellung von Qt for Embedded Linux und Qt/X11 (Quelle: Trolltech)

Das Fenstersystem ist in einer Client-Server-Architektur strukturiert. Der QWS-Server ist dabei für die Verwaltung des Framebuffers und der Eingabegeräte zuständig, d.h. er sorgt für die kombinierte Darstellung der Fenster aller Qt-Anwendungen auf dem Bildschirm und leitet alle Eingabeereignisse im Bereich eines bestimmten Fensters an den jeweils dazugehörigen QWS-Client, also eine bestimmte Qt-Anwendung, weiter. Die Qt-Anwendungen ihrerseits fordern die Speicherbereiche zum Zeichnen ihrer Fenster vom QWS-Server an und informieren den Server anschließend über alle Änderungen, die sie am Fensterinhalt vorgenommen haben. Der Server berechnet daraufhin die Überlappungen der einzelnen Fenster und kopiert die sichtbaren Anteile in den Framebuffer. Zur Optimierung der Darstellungsgeschwindigkeit und des Speicherbedarfs besteht aber auch die Möglichkeit, dass Clients ohne Umweg über den Server und ohne vorheriges Zeichnen in einen Backing Store direkt in den Framebuffer zeichnen. Dies wird dadurch ermöglicht, dass jeder Client den kompletten Framebuffer in seinen Adressraum gemappt bekommt. Auf diese Weise wird auch die von Qt angebotene „Fensterschnappschuss“-Funktionalität (`QPixmap::grabWindow()`) realisiert, ein gemeinsamer Zugriff auf den Framebuffer stellt allerdings auch ein erhebliches Sicherheitsrisiko dar.

Anders als bei anderen Fenstersystemen ist der QWS-Server kein eigenständiger Prozess, sondern eine zusätzliche Funktion, die von einem der Clients übernommen wird. Das könnte z.B. eine Anwendung sein, die über eine Eingabezeile oder klickbare Icons das Starten weiterer Qt-Anwendungen ermöglicht, welche sich anschließend als QWS-Clients mit der Server-Anwendung verbinden. Prinzipiell ist aber jede Qt-Anwendung als QWS-Server geeignet. Dies wird dadurch ermöglicht, dass der komplette Server-Code in der QtGui-Bibliothek enthalten ist, die von jeder grafischen Qt-Anwendung statisch oder dynamisch eingebunden wird. Es besteht auch die Möglichkeit, mehrere Anwendungen als QWS-Server zu starten, wenn mehrere Ausgabegeräte zur Verfügung stehen.

Die Kommunikation zwischen dem QWS-Server und den externen QWS-Clients wird mittels Shared Memory (für die Bekanntmachung der Fensterpositionen und -größen) und UNIX Domain Sockets realisiert. Die Client-Anwendung, welche gleichzeitig die Serverfunktion übernimmt, kann dagegen über lokale Speicherreferenzen und Methodenaufrufe direkt mit dem Server-Teil kommunizieren, wodurch der Ressourcenbedarf für den Fall, dass nur eine einzige grafische Anwendung auf dem Zielgerät ausgeführt werden soll, nochmals reduziert wird.

Zur weiteren Senkung des Speicherbedarfs besteht auch die Möglichkeit, verschiedene Qt-Bestandteile von der Kompilierung der Bibliothek auszunehmen, wenn sie nicht benötigt werden. Von dieser Möglichkeit wird auch bei der Genode-Portierung Gebrauch gemacht, um auf nicht vom Genode-System unterstützte Bestandteile wie beispielsweise Audioausgabe oder Druckdialoge zu verzichten.

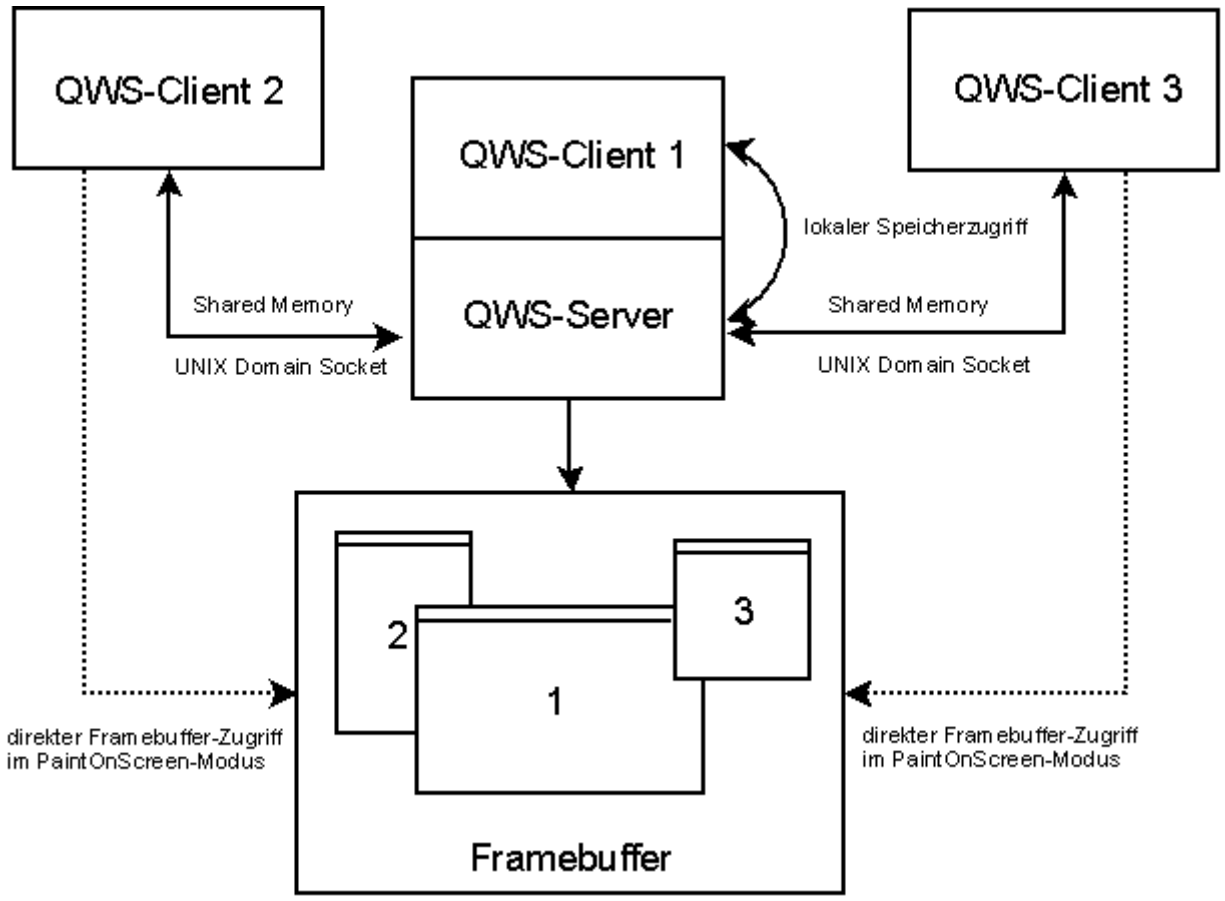

Abbildung 6: QWS Client-Server-Architektur

3 Verwandte Arbeiten

Im Folgenden sollen einige weitere GUI-Bibliotheken vorgestellt werden, die auf experimentellen, kleinen oder eingebetteten Betriebssystemen zum Einsatz kommen.

3.1 Qt 3 auf DROPS

Mit der Portierung[14] von Qt 3 auf das „Dresden Realtime Operating System“ (DROPS), das ebenfalls an der TU-Dresden entwickelt wird, konnte gezeigt werden, dass sich Qt/Embedded mit vertretbarem Aufwand auf ein neues Betriebssysteme portieren lässt und im Ergebnis eine komfortable Entwicklung von leistungsfähigen grafischen Anwendungen sowie die Möglichkeit der einfachen Portierung einer großen Anzahl bereits vorhandener Anwendungen auf DROPS ermöglicht. Diese positive Erkenntnis trug schließlich auch zur Motivation bei, Qt auch auf Genode zu portieren. Die auf DROPS portierte Qt-Bibliothek unterstützt den Zugriff auf Dateisystem und Netzwerk, Threads und Synchronisationsmechanismen, Benutzereingaben und grafische Ausgabe über die DROPS-Konsole l4con, den Window-Server DoPE oder die SDL-Bibliothek. Des Weiteren ist die Ausgabe von Sound über das „DROPS sound system“ möglich. Die Genode-Portierung beschränkt sich dagegen auf die Bereitstellung der GUI-Funktionalität. Für die grafische Ausgabe wird in der DROPS-Portierung das im Grundlagenkapitel angesprochene Shared-Framebuffer-Konzept unterstützt, wohingegen in der Genode-Portierung jede Qt-Anwendung als QWS-Server mit alleinigem Zugriff auf einen eigenen virtuellen Framebuffer ausgeführt wird.

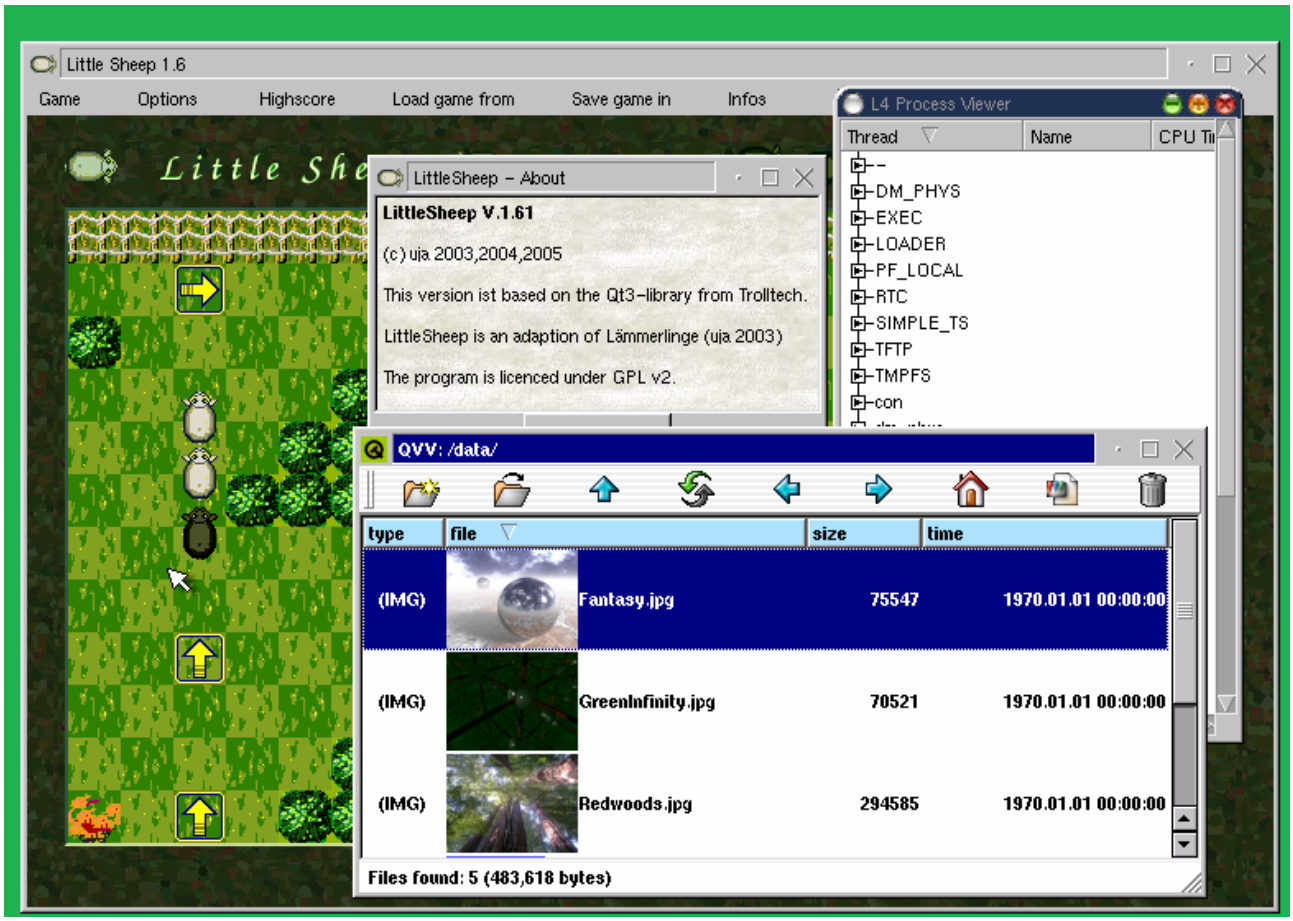

Abbildung 7: Qt 3 auf DROPS (Quelle: http://demo.tudos.org)

3.2 DOpE

Das „Desktop Operating Environment“ (DOpE)[15] ist ein Window-Server für Echtzeit- und eingebettete Systeme. Es wurde ursprünglich für DROPS entwickelt, unterstützt aber auf Grund seiner nur auf Threads und Adressräume begrenzten Abhängigkeit vom unterliegenden Betriebssystem z.B. auch Linux und sogar eingebettete Systeme ohne Betriebssystem. Eine Genode-Version existiert ebenfalls. Auf Echtzeitsystemen zeichnet sich der Server im Gegensatz zu Qt vor allem dadurch aus, dass er ausgewählten Clients eine feste Bildwiederholrate garantieren kann, während die übrigen Clients nach bestem Bemühen bedient werden. Dies wird unter Anderem dadurch ermöglicht, dass der Server das Zeichnen der Widgets für die Benutzerinteraktion selbst übernimmt und vom Client nur eine textuelle Beschreibung der Widgetanordnung aus einem gemeinsam genutzten Speicherbereich entnimmt. Dadurch kann der Server ohne aktive Mitarbeit der einzelnen Clients und somit unabhängig von deren Antwortzeiten eine optimale Reihenfolge der einzelnen Zeichenoperationen berechnen. In der Genode-Version ist der Server als Bibliothek ausgeführt, die jeweils gegen eine Client-Applikation gelinkt wird. Um dennoch mehrere DoPE-Applikationen gleichzeitig nutzen zu können, erfolgt die Grafikausgabe der einzelnen Server nicht direkt auf der Grafikhardware, sondern über den Nitpicker GUI-Server, der mehrere Fenstersysteme auf dem selben Bildschirm vereint darstellen kann. Auf die Funktionsweise von Nitpicker wird in Kapitel 4.5.1 noch genauer eingegangen.

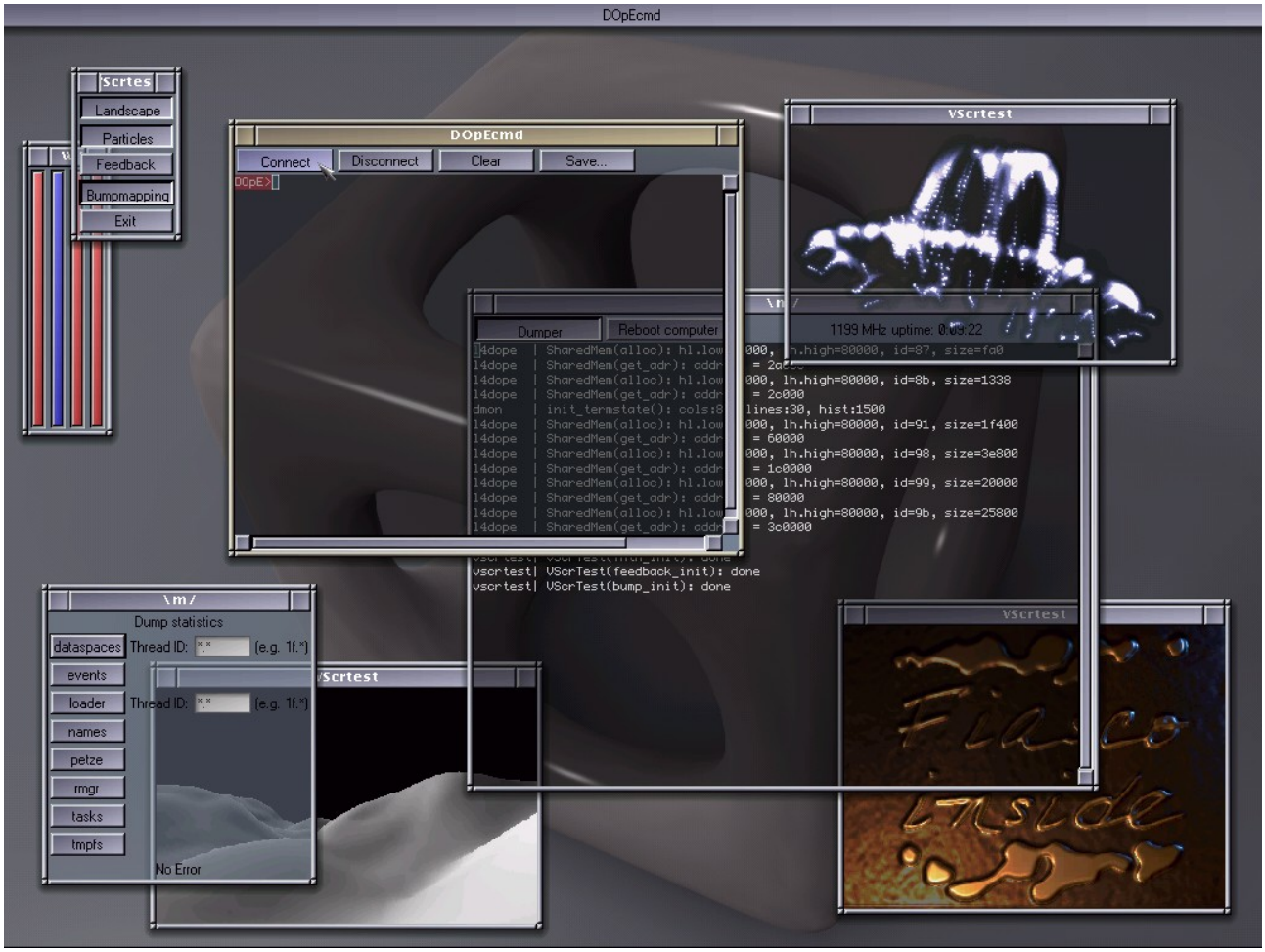

Abbildung 8: DOpE (Quelle: Genode Labs)

3.3 Scout Widgets

Der „Scout Tutorialbrowser“[16] ist eine der wenigen Anwendungen, die bereits auf Genode portiert wurden. Er ermöglicht das Lesen und Navigieren innerhalb mehrseitiger Hypertext-Dokumente und bringt dafür eine kleine Auswahl an Widgets mit. Da diese in einer eigenen Bibliothek zusammengefasst sind, können sie auch in anderen Anwendungen verwendet werden. Ein Beispiel dafür ist die „Launchpad“-Anwendung, die eine grafische Schnittstelle zum Starten und Beenden von ausgewählten Genode-Anwendungen mit individueller Konfiguration der jeweiligen RAM-Quota anbietet. Der Einsatz der Scout Widgets-Bibliothek bietet sich wegen des kleineren Umfanges im Vergleich zu Qt vor allem für Anwendungen an, die großen Wert auf eine möglichst kleine Trusted Computing Base legen.

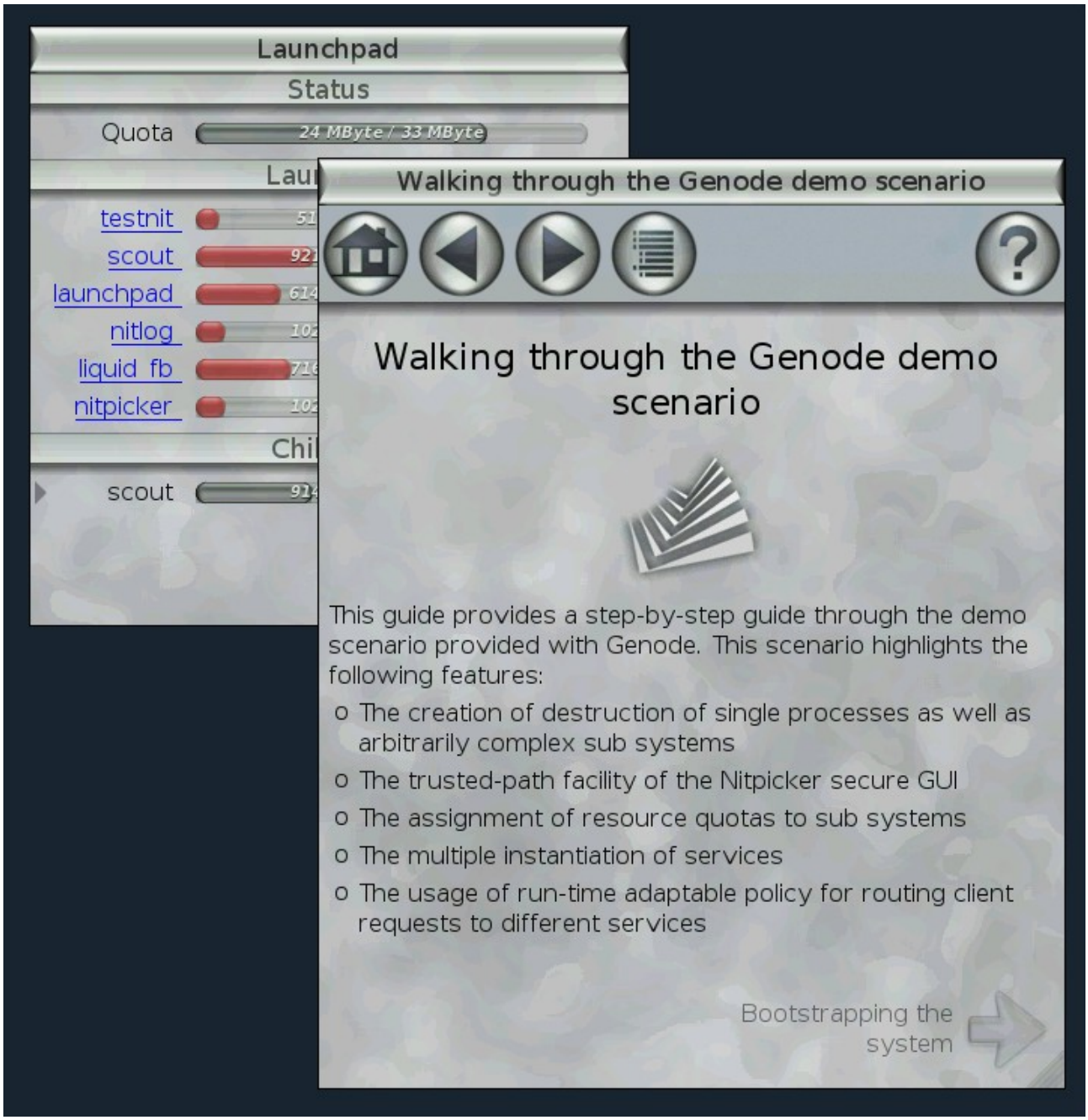

Abbildung 9: Scout Widgets: Launchpad und Scout Tutorialbrowser

3.4 Xynth Windowing System

Das „Xynth Windowing System“[17] ist ein in ANSI C programmiertes portables Fenstersystem, das auch besonders für eingebettete Systeme geeignet ist. Es zeichnet sich durch eine Client-Server-Struktur aus, wobei die Kommunikation zwischen Client und Server über TCP/IP, Unix Domain Sockets oder einen mitgelieferten minimalen Socket-Stack erfolgen kann. Der Server und alle Clients greifen direkt auf den Videospeicher zu, wodurch sich ein geringer Speicher- und CPU-Verbrauch ergibt. Das System enthält auch einen minimalen Fenstermanager und ein minimales Widget-Set. Darüber hinaus sind Portierungen der Bibliotheken SDL[18] und GTK+[19] vorhanden. Die Architektur von Xynth weist einige Ähnlichkeiten mit dem QWS-System in „Qt for Embedded Linux“ auf, für Genode ist ein C++-Framework wie Qt allerdings etwas besser geeignet und ein direkter Zugriff aller Anwendungen auf den Videospeicher ist aus Sicherheitsgründen eher nicht gewünscht.

Abbildung 10: Xynth Windowing System (Quelle: http://www.xynth.org)

4 Portierung

4.1 Vorbetrachtungen und Entwurfsziele

Zu Beginn der Portierung stellte sich die Frage, welche Qt-Bestandteile sich überhaupt mit der gegebenen Genode-Infrastruktur portieren lassen würden. Das Genode-Basissystem bietet Unterstützung für Threads, Adressräume, Speicherallokation, Shared Memory, Locks und Semaphore und es steht ein Timer-Service zur Verfügung. Damit waren bis auf die Verfügbarkeit eines POSIX-artigen Dateisystems die wichtigsten Systemvoraussetzungen für die Portierung der QtCore-Bibliothek erfüllt. Für Qt-Anwendungen, die auf externe Ressourcen wie Hilfetexte oder Bilddateien angewiesen sind, besteht zumindest die Möglichkeit, diese Dateien mit Hilfe des Resource Compilers `rcc` als statische Ressourcen mit in die ausführbare Anwendung zu linken und der Anwendung über einen speziellen Dateipfad zugänglich zu machen.

Für die QtGui-Bibliothek werden in der für die Portierung vorgesehenen Qt-Variante „Qt for Embedded Linux“ zusätzlich zu den Systemvoraussetzungen der QtCore-Bibliothek noch ein Framebuffer für die Grafikausgabe und Möglichkeiten zur Erkennung von Tastatur- und Mauseingaben benötigt. Diese Bedingung ist durch entsprechende Genode-Services, im Speziellen durch den Nitpicker GUI-Server, ebenfalls erfüllt.

Damit standen also zumindest die benötigten Basisdienste für die Portierung des GUI-Teils von Qt grundsätzlich in Genode zur Verfügung. Außer den Basisdiensten werden aber auch noch verschiedene systemunabhängige Funktionen aus der C-Bibliothek (wie z.B. strcpy()) benötigt, welche auf Genode noch nicht zur Verfügung steht. Die C-Bibliothek wird von Qt auch als Schnittstelle zu den Systemfunktionen verwendet, so dass an dieser Stelle einiger Portierungs- und Anpassungsbedarf bestehen würde.

Die weiteren Qt-Module bauen größtenteils auf dem QtCore-Modul auf, benötigen aber darüber hinaus noch weitere Funktionen aus der C-Bibliothek, die für QtCore und QtGui nicht benötigt werden. Für das QtNetwork-Modul wäre zusätzlich noch die Erweiterung der Genode-Infrastruktur um Netzwerkkartentreiber und Netzwerkprotokollstacks notwendig. Für die Belegarbeit war allerdings nur die Portierung der QtCore- und QtGui-Module relevant.

Die Portierungsarbeit würde sich also aus drei Teilen zusammensetzen:

- Integration des Qt-Sourcecodes in das Genode-Buildsystem
- Anpassung der Schnittstellen zwischen Qt und den Genode-Systemdiensten (und falls nötig eine Erweiterung der Genode-Systemdienste um möglicherweise fehlende Funktionalität)
- Bereitstellung der von Qt benötigten systemunabhängigen Funktionalität, die unter Embedded Linux von der C-Bibliothek bereitgestellt wird

Dabei war abzusehen, dass es mehrere Möglichkeiten der Umsetzung geben würde. Als grundlegende Einschränkung konnte bei der Designentscheidung für eine bestimmte Möglichkeit gelten, dass das anwendungsseitige Qt-API nicht verändert werden darf, denn dadurch wäre die Kompatibilität mit existierenden Qt-Anwendungen nur noch eingeschränkt vorhanden. Weiterhin galt es bei der Entscheidungsfindung, die Besonderheiten des Genode-Systems zu beachten. Ein wichtiges Ziel der Genode-Architektur ist es, die Trusted Computing Base der einzelnen Anwendungen jeweils möglichst klein zu halten. Das bedeutet unter Anderem, dass das Genode-Framework möglichst wenig von Funktionen des unterliegenden Systems (Betriebssystemkern bzw. Linux-Syscalls) abhängen sollte, aber auch, dass die einzelnen Klassen des Frameworks eine möglichst geringe Komplexität aufweisen und nicht mit Features überladen werden, die sich auch in einer separaten Klasse unterbringen lassen würden. Letzteres Kriterium wäre vor allem dann zu

beachten, wenn Erweiterungen des Basissystems für die vollständige und korrekte Portierung erforderlich sein würden. Ein weiterer Grund dafür, die Abhängigkeiten vom unterliegenden System möglichst gering zu halten, besteht darin, dass manche der zukünftig von Genode unterstützten Systeme die benötigte Funktionalität möglicherweise gar nicht oder nur unvollständig oder mit einer unterschiedlichen Semantik bereitstellen.

4.2 Integration in das Genode-Buildsystem

Der erste Schritt der Portierung bestand darin, den Qt-Quellcode in das Genode-Buildsystem zu integrieren. Charakteristisch für Qt ist der Einsatz des qmake-Programms, welches aus einer speziellen deklarativen Projektdatei ein plattformspezifisches Makefile erzeugt. Unter Embedded Linux ist dies beispielsweise ein Makefile für GNU `make`. Das Genode-Buildsystem verwendet allerdings ein eigenes Buildsystem, das zwar auch auf GNU `make` basiert, aber mit den Makefiles für Embedded Linux nichts anfangen kann. Um die Qt-Bibliotheken unter Genode kompilieren zu können, wurden entsprechende Genode-Builddateien von Hand erstellt. Dieser Vorgang war recht mühsam und musste nach Änderungen an der Qt-Konfiguration oft wiederholt werden, allerdings hat sich dabei auch angedeutet, dass eine automatisierte Generierung dieser Builddateien mit Hilfe eines Scriptes mit großer Wahrscheinlichkeit möglich ist. Als Alternative zur Erstellung von vollständigen Builddateien für das Genode-Buildsystem hätte theoretisch auch die Möglichkeit bestanden, das Qt-Buildsystem so anzupassen, dass es die Bibliotheks-Binaries für die Genode-Plattform direkt, also unabhängig vom Genode-Buildprozess, erzeugt. Dies hätte aber unter Anderem zu Problemen mit der Abhängigkeitsbehandlung geführt – wenn eine Genode-Bibliothek, die von Qt benötigt wird, noch nicht gebaut wurde, müsste dieser Buildvorgang auf irgendeine Weise aus dem Qt-Buildsystem heraus angestoßen werden, wohingegen sich das Genode-Buildsystem bei der gewählten Lösung mit den vollständigen Genode-Builddateien automatisch um den Bau der benötigten Bibliotheken kümmert.

Vor dem eigentlichen Kompilieren der Qt-Bibliotheken steht noch der Konfigurationsvorgang, der durch den Aufruf des `configure`-Scriptes gestartet wird. Dabei werden in Abhängigkeit von den übergebenen Konfigurationsparametern und den Definitionen in einer speziellen Konfigurationsdatei, die den gewünschten Funktionsumfang der Qt-Bibliotheken festlegen, einige von Qt benötigte Header-Dateien erstellt sowie die zum Erstellen der Qt-Bibliotheken und von Qt-Anwendungen benötigten Hilfsprogramme `moc`, `rcc` und `uic`. Dieser Konfigurationsvorgang muss auch vor dem Start des Genode-Buildvorganges durch einmaligen Aufruf des Konfigurationsscriptes durchgeführt werden.

Qt-Anwendungen werden üblicherweise ebenfalls mit Hilfe des `qmake`-Programms aus einer entsprechenden Projektdatei erzeugt. Der Inhalt einer solchen Projektdatei besteht größtenteils aus der Belegung von Variablen mit Werten, in einer Syntax, die mit der Syntax von GNU Makefiles viele Gemeinsamkeiten hat. Da die Builddateien für Genode-Anwendungen teilweise ähnlich aufgebaut sind und Qt-Anwendungen im Gegensatz zu den umfangreichen Qt-Bibliotheken oftmals nur eine einzige `qmake`-Projektdatei benötigen, war es hier möglich, die Genode-Builddateien so zu gestalten, dass eine während des Buildvorganges eingelesene, einfache `qmake`-Projektdatei so weit ausgewertet werden kann, dass alle nötigen Informationen ohne manuelle Nachbearbeitung extrahiert werden können. Auf diese Weise ist es möglich, einfache Qt-Anwendungen ähnlich komfortabel wie auf den anderen unterstützten Plattformen zu entwickeln, indem man zuerst alle anwendungsspezifischen Quellcode-, Qt Designer- und Ressourcendateien erstellt und anschließend durch einen einfachen Aufruf von „`qmake -project`“ automatisch die dazu passende `qmake`-Projektdatei erstellen lässt. Die Genode-Builddateien befinden sich alle in einem Template-Verzeichnis, das als Kopiervorlage für neue Anwendungen dienen kann. Eine manuelle Nachbearbeitung der Builddateien ist dabei möglich, aber bei vielen Anwendungen nicht nötig.

In der Zusammenstellung der Template-Builddateien ist auch einer der von Qt mitgelieferten Fonts enthalten und wird bei grafischen Qt-Anwendungen als Ressource in die ausführbare Datei gelinkt. Diese Vorgehensweise ist notwendig, da noch kein Dateisystem existiert, aus denen der Font sonst zur Laufzeit geladen werden könnte.

```
# ---------------
# qt_launchpad.pro
# ----------------

TEMPLATE = app

TARGET = qt_launchpad          # Name der Anwendung

QT += core \                   # benötigte Qt-Module
      gui

HEADERS += kbyte_loadbar.h \   # Header mit Qt-Makros aus denen moc Meta-Code
           child_entry.h \     # erzeugt
           launch_entry.h \
           qt_launchpad.h

SOURCES += kbyte_loadbar.cpp \ # Dateien mit Anwendungs-Code
           child_entry.cpp \
           launch_entry.cpp \
           main.cpp \
           qt_launchpad.cpp

FORMS += child_entry.ui \      # GUI-Beschreibungsdateien, aus denen mit moc
         launch_entry.ui \     # C++-Header-Dateien erzeugt werden
         qt_launchpad.ui

RESOURCES += font.qrc          # Ressourcen, die zur Anwendung gelinkt werden
```

Listing 1: einfache `qmake`-Projektdatei

4.3 C- und C++-Support

„Qt for Embedded Linux“ wurde, wie der Name schon vermuten lässt, eigentlich für Embedded Linux konzipiert und setzt deshalb eine POSIX-Umgebung mit dazugehöriger C-Bibliothek voraus. Für Genode existiert so eine C-Bibliothek jedoch bisher nicht. Zur Lösung dieses Problems standen verschiedene Optionen zur Auswahl. Die Portierung einer vollständigen C-Bibliothek, wie z.B. der uClibc, wäre sicher im Hinblick auf die sich dadurch ergebenden Portierungsmöglichkeiten weiterer Anwendungen aus der UNIX-Welt besonders wünschenswert. Andererseits ist der Funktionsumfang einer solchen C-Bibliothek sehr groß und für einen großen Teil der systemgebundenen Funktionalität der Bibliothek müsste erst einmal eine entsprechende Grundlage in Genode geschaffen werden. Hier bestand die Gefahr, dass die eigentliche Aufgabe dieser Belegarbeit, also die Portierung von Qt auf Genode, zu Gunsten der Portierung einer C-Bibliothek auf Genode zu kurz gekommen wäre, da bereits der Aufwand für die Qt-Portierung im Vorfeld schwer abzuschätzen war. Das Gegenteil zur Portierung einer vollständigen C-Bibliothek wäre der Versuch, den Qt-Sourcecode direkt an die vorhandenen Genode-Schnittstellen anzupassen bzw. die Teile auszukommentieren, für die keine passenden Genode-Schnittstellen vorhanden sind, um so komplett ohne eine C-Bibliothek auszukommen. Dies wäre zwar prinzipiell möglich, würde aber auf Grund der noch recht begrenzten Genode-Funktionalität doch zu vielen auskommentierten Stellen führen, die Updates auf neuere Qt-Versionen erschweren würden. Darüber hinaus wäre mit dieser Alternative in jedem Fall eine manuelle Nachbearbeitung des Qt-Sourcecodes nötig, sobald eine heute noch fehlende Funktionalität zukünftig von Genode angeboten werden sollte. Für den nicht unwahrscheinlichen Fall, dass irgendwann einmal doch eine C-Bibliothek auf Genode portiert wird, wäre die Notwendigkeit dieser manuellen Nachbearbeitungen bzw. der Rückgängigmachung der zuvor mangels C-Bibliothek durchgeführten Änderungen natürlich besonders unerfreulich. Für die nicht systemgebundene Funktionalität, wie mathematische Funktionen oder Funktionen zur Stringverarbeitung wäre zudem immer noch die Implementierung in einer externen Bibliothek angeraten, um die Duplizierung von Code zu vermeiden.

Aus diesen Gründen schien es schließlich am sinnvollsten, möglichst wenige Änderungen am Qt-Sourcecode vorzunehmen und stattdessen die von Qt benötigten C-Funktionen in einer provisorischen C-Bibliothek `qt_c` bereitzustellen, die im Falle einer zukünftigen Portierung einer vollständigen C-Bibliothek problemlos durch diese ersetzt werden kann. Dabei wurden zuerst nur die Funktionsrümpfe ohne eigentliche Implementierung erzeugt, um Qt überhaupt linken zu können. Anschließend wurden die Funktionen, die zum Betrieb von einfachen Qt-Anwendungen unbedingt erforderlich sind, entweder neu implementiert, falls sie Genode-spezifische Funktionalität voraussetzten, oder anderenfalls die Implementierung teilweise oder ganz aus der uClibc oder der speziell für grundlegende Genode-Basisdienste entwickelten und im Framework enthaltenen `mini_c`-Bibliothek übernommen. Die neu implementierten, Genode-spezifischen Funktionen könnten bei einer zukünftigen Portierung einer vollständigen C-Bibliothek natürlich teilweise oder komplett in diese übernommen werden.

Von der Standard-C++-Bibliothek wird in den QtCore- und QtGui-Modulen nur wenig Gebrauch gemacht. Hier war lediglich die Implementierung der `new`- und `delete`-Operatoren zur Allokierung und Freigabe von Speicher auf dem Heap notwendig, da die von Genode bereitgestellte Version des `new`-Operators für die Speicherreservierung auf ein zusätzlich zu übergebendes Allokator-Objekt zurückgreift und der Standard-`delete`-Operator nur eine Fehlermeldung ausgibt. Die neu implementierten Versionen für Qt verwenden dagegen die `malloc()`- und `free()`-Funktionen aus der C-Support-Bibliothek, die ihrerseits auf den Genode-Heap-Allokator zurückgreifen.

4.4 Portierung der QtCore-Bibliothek

4.4.1 Timer

Timer ermöglichen es, bestimmte Aktionen zeitversetzt oder regelmäßig, in periodischen Abständen durchzuführen. In Qt gibt es grundsätzlich zwei Möglichkeiten, Timer zu verwenden. Die erste Möglichkeit wird von der Klasse `QObject`, die die Basisklasse der meisten Qt-Klassen ist, angeboten. Durch Aufruf der Methode `startTimer()` wird periodisch nach Ablauf des übergebenen Intervalls die virtuelle Methode `timerEvent()` aufgerufen, in welcher die gewünschte Aktion implementiert werden kann. Durch Aufruf der Methode `killTimer()` kann der Timer wieder gestoppt werden. Die zweite Möglichkeit besteht in der Verwendung der Klasse `QTimer`. Im Gegensatz zur Timer-Funktionalität der Klasse `QObject` wird hier nach Ablauf des Intervalls keine fest definierte Methode innerhalb des selben Objektes aufgerufen, sondern ein `timeout()`-Signal emittiert, das mit Hilfe des Signal-Slot-Mechanismus' mit beliebigen Slot-Methoden, die auch zu anderen Objekten gehören können, verknüpft werden kann.

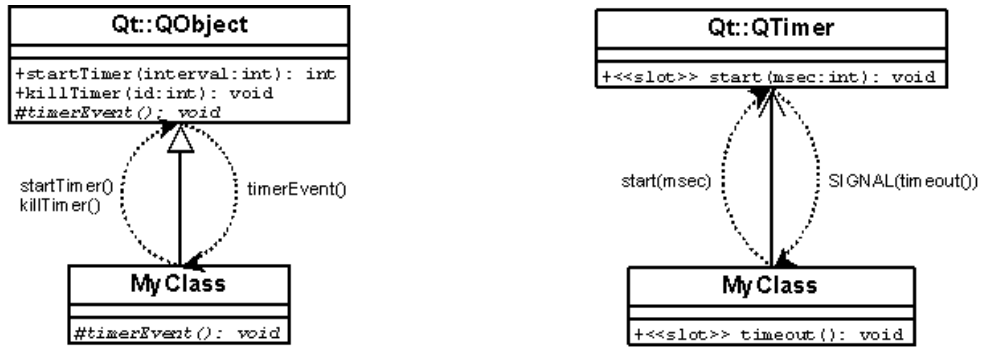

Abbildung 11: Timer in Qt

Intern wird die Timer-Funktionalität einheitlich für beide Nutzungsmöglichkeiten in einer von der abstrakten Klasse `QAbstractEventDispatcher` abgeleiteten plattformspezifischen Klasse realisiert, die sich neben Timern auch um die Behandlung von Sockets kümmert. Für Embedded Linux ist dies die Klasse `QEventDispatcherUNIX`. In dieser wird das Warten auf Socket- und Timerereignisse in einem einzigen `select()`-Aufruf kombiniert, indem der `select()`-Funktion die Zeit bis zum Ablauf des nächstfälligen Timers als Timeout-Parameter übergeben wird. Der Aufruf blockiert dann so lange, bis entweder an einem der überwachten Dateideskriptoren Daten bereitstehen oder der Timeout abgelaufen ist. Zu den überwachten Dateideskriptoren gehört dabei auch immer das Ende einer lokalen Pipe. Dadurch ist es für einen anderen Thread möglich, den Blockiervorgang vorzeitig zu unterbrechen, indem dieser die Methode `wakeUp()` aufruft, die ihrerseits ein Zeichen in die lokale Pipe schreibt, wodurch der `select()`-Aufruf schließlich zurückkehrt. Nach der Rückkehr aus `select()` wird durch einen Abgleich mit der aktuellen Uhrzeit überprüft, ob und welche Timer inzwischen abgelaufen sind und gegebenenfalls das entsprechende Qt-Timer-Ereignis ausgelöst. Anschließend wird wieder der nächstfällige Timer ermittelt und ein neuer `select()`-Aufruf gestartet.

Da unter Genode weder Socket-Funktionalität noch eine Möglichkeit zur Ermittlung der aktuellen Uhrzeit gegeben sind, wurde die Timerbehandlung in der für die Genode-Plattform abgeleiteten Klasse `QEventDispatcherGenode` mit Hilfe von Threads realisiert. Dabei wird für jeden registrierten Timer ein Thread erzeugt, der das gegebene Zeitintervall unter Verwendung des Genode-Timer-Services abwartet und anschließend eine Semaphore erhöht, an der der `main()`-Thread ähnlich wie beim `select()`-Aufruf unter Linux blockiert. Der `main()`-Thread ermittelt daraufhin, welcher der Threads fertig ist, generiert gegebenenfalls das entsprechende Qt-Timer-

Ereignis, startet dann den Timer-Thread für das nächste Intervall neu und wartet anschließend wieder an der Semaphore auf die Beendigung des nächsten Timer-Threads bzw. auf eine explizite Unterbrechung der Blockierung durch einen anderen Thread, die mittels einer Erhöhung der Semaphore in der `wakeUp()`-Methode auch bei der Genode-Implementierung gewährleistet ist.

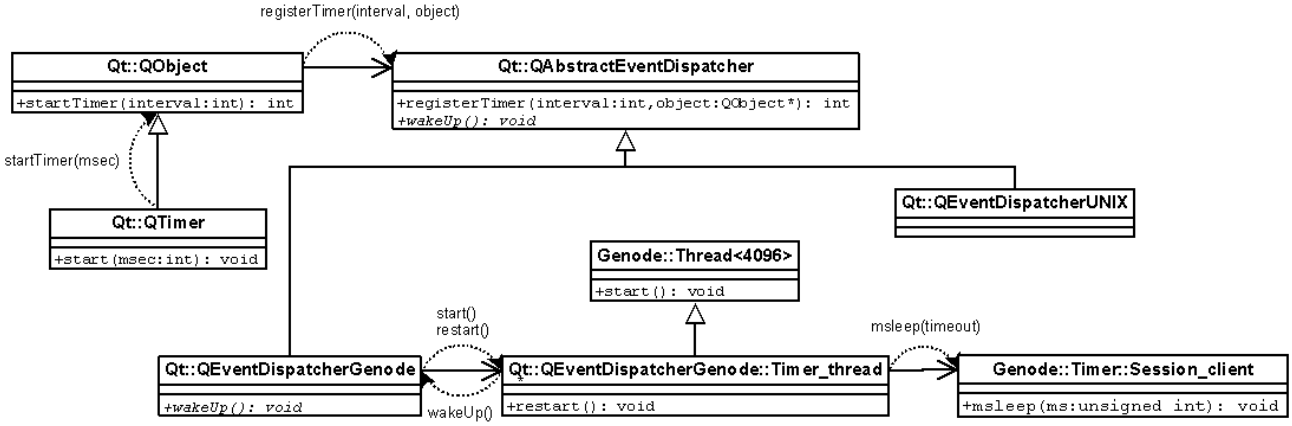

Abbildung 12: Timer-Implementierung

4.4.2 Synchronisation

Bei der Arbeit mit mehreren Threads ist es wichtig, kritische Bereiche vor gleichzeitigem Zugriff zu schützen, um inkonsistente Zustände zu vermeiden. Dies kann durch den Einsatz von Mutexes, Semaphoren und Konditionsvariablen gewährleistet werden. In Qt stehen diese drei Mechanismen plattformunabhängig mit den Klassen `QMutex`, `QSemaphore` und `QWaitCondition` zur Verfügung.

Die Klasse `QSemaphore` greift intern auf die Klassen `QMutex` und `QWaitCondition` zurück und musste deshalb nicht für Genode portiert werden. Für die Klassen `QMutex` und `QWaitCondition` sind jedoch plattformspezifische Implementierungen notwendig. Unter Embedded Linux wird dabei auf Funktionen der Pthreads-Bibliothek wie z.B. `pthread_mutex_lock()` und `pthread_cond_wait()` zurückgegriffen. Diese Funktionen sind unter Genode in dieser Form nicht gegeben, es stand aber mit den Genode-Klassen `Lock` und `Semaphore` eine vergleichbare Funktionalität zur Verfügung, mit der sich die Klassen `QMutex` und `QWaitCondition` schließlich realisieren ließen. Für die erfolgreiche Portierung war es dabei allerdings notwendig, die ursprüngliche Genode-Semaphore-Klasse zu erweitern, da sie bisher nur auf ein spezielles Anwendungsszenario mit einem Producer und einem Consumer ausgelegt war. Die Erweiterung bestand neben der Anpassung der Zählerauswertung hauptsächlich darin, für die Blockierung eines Threads in der `down()`-Methode jeweils einen lokalen Lock zu verwenden und diesen in einer (durch einen Lock geschützten) Liste einzutragen, aus welcher die `up()`-Methode später nach Erhöhung des Zählerwertes den nächsten Lock auswählt und diesen anschließend freigibt, um die Blockierung des dazugehörigen Threads in der `down()`-Methode aufzuheben. Darüber hinaus wurde zum Erreichen einer vollständigen Portierung noch eine Möglichkeit benötigt, ein eventuelles Blockieren des `down()`-Aufrufes nach Ablauf eines benutzerdefinierten Timeouts zu unterbrechen. Für die Realisierung dieser Möglichkeit standen mehrere Alternativen zur Auswahl. Die nächstliegende Lösung hätte darin bestanden, die `Semaphore`-Klasse einfach um eine solche `down_timed()`-Methode mit Timeout-Parameter zu erweitern und für deren Implementierung von den angebotenen Möglichkeiten des unterliegenden Betriebssystemkernes bzw. dessen Support-Bibliotheken Gebrauch zu machen. Allerdings galt es bei der Designentscheidung für die Art und Weise der Realisierung auch, die Genode-Philosophie der minimalen Komplexität des Frameworks und der minimalen Abhängigkeit vom unterliegenden

System zu berücksichtigen, wonach eine Implementierung, die die gewünschte Funktionalität unter möglichst ausschließlicher Verwendung von bereits gegebenen Genode-Mitteln anbieten kann, einer Implementierung mit Zugriff auf das unterliegende System grundsätzlich vorzuziehen wäre. Tatsächlich lässt sich die benötigte Timeout-Semantik dadurch realisieren, dass vor jeder Blockierung innerhalb der `down_timed()`-Methode ein Watchdog-Thread erstellt wird, der die Aufgabe erhält, den entsprechenden lokalen Lock nach Ablauf des Timeouts wieder freizugeben und damit die Blockierung vorzeitig aufzuheben. Das ist der gleiche Mechanismus, der bereits für die Timer-Implementierung verwendet wurde. Diese Variante war also in diesem Fall die bessere Wahl. Weiterhin stellte sich die Frage, ob es nicht auch noch eine bessere Alternative zur direkten Erweiterung der Semaphore-Klasse gäbe. Da die Timeout-Semantik von vielen Anwendungen gar nicht benötigt wird und die Implementierung der `down_timed()`-Methode unabhängig von den Implementierungen der anderen Methoden der Semaphore-Klasse ist, bot es sich hier vielmehr an, diese Funktionalität in eine separate, von der Klasse `Semaphore` abgeleitete Klasse `Timed_semaphore` auszulagern. Eine Implementierung innerhalb von Qt wäre ebenfalls möglich gewesen. Da eine Semaphore mit Timeout-Semantik aber eventuell auch für andere Genode-Anwendungen nützlich sein könnte, fiel die Wahl in diesem Fall auf die genannte `Timed_semaphore`-Klasse im Genode-Namespace.

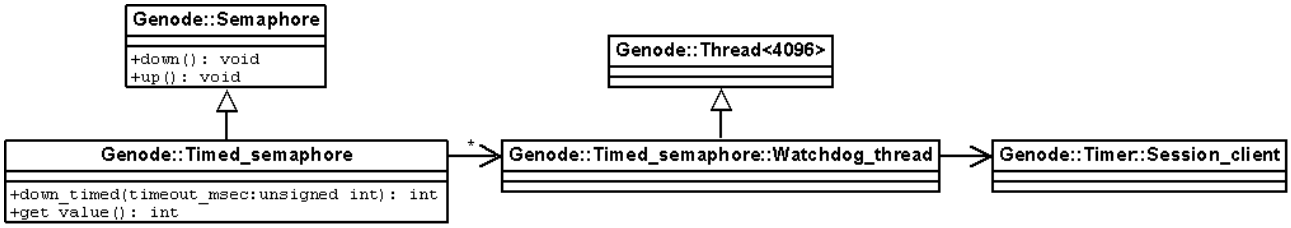

Abbildung 13: Semaphor-Implementierung

4.4.3 Threads

Für den Einsatz von Threads steht in Qt die Klasse `QThread` zur Verfügung. Wie bei den Synchronisationsmechanismen greift auch hier die interne Implementierung für Embedded Linux auf Funktionen der Pthreads-Bibliothek zurück, die unter Genode nicht verfügbar ist. Als Schnittstelle für die Thread-Funktionalität wird von Genode dagegen die Klasse `Thread` angeboten. Allerdings wird von dieser Klasse nur ein Teil der Funktionalität, die von der Klasse `QThread` benötigt wird, bereitgestellt. Es fehlen konkret Möglichkeiten, die Stackgröße eines neuen Threads zur Programmlaufzeit zu konfigurieren, die Identität (Thread-ID) des momentan ausgeführten Threads zu ermitteln sowie Daten Thread-lokal zu speichern. Um die Klasse `QThread` vollständig auf Genode portieren zu können, war es deshalb erforderlich, diese fehlende Funktionalität zu implementieren. Bei der Entscheidungsfindung über die Art der Realisierung galt es auch hier wieder, die bereits erwähnte Genode-Philosophie der minimalen Komplexität des Frameworks und der minimalen Abhängigkeit vom unterliegenden System zu berücksichtigen. Die Möglichkeit, die Stackgröße eines neuen Threads dynamisch zur Programmlaufzeit festlegen zu können, ist sicher auch für andere Genode-Anwendungen wünschenswert. Aus diesem Grund schien hier die Erstellung einer neuen Klasse `Thread_qt` für Threads mit konfigurierbarer Stack-Größe die geeignete Wahl. Diese Klasse wurde, wie auch die Klasse `Thread`, von der Klasse `Thread_base` abgeleitet und um eine Methode `set_stack_size()` erweitert, mit der die gewünschte Stack-Größe festlegt werden kann. In der Implementierung für Genode/Fiasco wird dabei auch gleichzeitig der Speicher für den Stack dynamisch allokiert. In der Implementierung für Genode/Linux hat die Methode dagegen keine praktische Auswirkung, da hier die Stacks vollständig von der Pthreads-Bibliothek verwaltet werden.

Für die Ermittlung einer eindeutigen Thread-ID zur Identifizierung des gerade ausgeführten Threads steht sowohl unter Linux als auch unter Fiasco eine entsprechende Systemfunktion bereit, die prinzipiell in einer einheitlichen Methode in der Klasse `Thread_qt` gekapselt werden könnte. Allerdings ist die Abhängigkeit von dieser Systemfunktion etwas problematisch, da sie möglicherweise auf zukünftigen (z.B. Capability-basierten) Systemen oder bei Nutzung von User-Level-Threads nicht verfügbar ist. Darum war es in diesem Fall besonders wünschenswert, eine alternative Möglichkeit zu finden, die ohne diese Abhängigkeit vom unterliegenden System auskommt. Tatsächlich konnte eine solche alternative Möglichkeit gefunden werden. Sie macht sich den Umstand zu Nutze, dass jeder Thread einen eigenen, unabhängigen Stack-Bereich im Adressraum des dazugehörigen Prozesses besitzt. Die eindeutige Identifizierung eines Threads ist nun möglich, wenn man feststellen kann, zu welchem Thread der Stack-Bereich gehört, in welchem sich die Adresse einer lokalen Variable, die ja üblicherweise auf dem Stack gespeichert wird, befindet. Diese Zuordnung wird in der Klasse `Thread_qt` mit Hilfe eines statischen Avl-Baumes vorgenommen, der für jeden neu erstellten Thread die Start- und Endadressen des dazugehörigen Stacks speichert. Beim Aufruf der statischen Methode `current_thread_id()` in der Klasse `Thread_qt` wird nun derjenige Eintrag im Avl-Baum gesucht, bei dem die Adresse der lokalen Lock-Guard-Variable[2], die den Zugriff auf den Avl-Baum während der Abarbeitung der Methode absichert, innerhalb der gespeicherten Bereichsgrenzen liegt. Als Thread-ID wird anschließend die Startadresse des gefundenen Stacks zurückgegeben oder 0, falls kein passender Eintrag gefunden wurde. Letzterer Fall tritt auf, wenn es sich bei dem gerade ausgeführten Thread um den `main()`-Thread handelt, da dieser nicht mit Hilfe der `Thread_qt`-Klasse erstellt wurde und somit keinen Eintrag im Avl-Baum besitzt. Somit wird dem `main()`-Thread immer die Thread-ID 0 zugeordnet.

Da der Stack eines neuen Threads unter Genode/Linux von der Pthreads-Bibliothek verwaltet wird, konnte die beschriebene Lösung in der Linux-spezifischen Implementierung der Klasse `Thread_qt` nicht angewandt werden. Hier wird deshalb ausnahmsweise auf die entsprechende Systemfunktion `pthread_self()` zurückgegriffen, was aber auch nicht tragisch ist, da die Linux-Version von Genode nur zu Entwicklungs- und Debugging-Zwecken dient.

Die Unterstützung für TLS wurde auf Wunsch der Genode-Autoren nicht in der Klasse `Thread_qt` oder einer anderen Klasse im Genode-Namespace implementiert, da dieses Feature zu sehr an der Sprache vorbeigeht und deshalb besser nicht verwendet werden sollte. Da das Qt-API aber TLS-Funktionalität anbietet, wurde diese innerhalb von Qt in der Klasse `QthreadPrivate` mittels einer klassenstatischen Hash-Map realisiert, die jeweils die Thread-ID des gerade ablaufenden Threads auf einen Zeiger abbildet, der auf die dazugehörigen Thread-lokalen Daten verweist.

2 bekommt im Konstruktor einen Lock übergeben und sperrt diesen; wenn das Lock-Guard-Objekt zerstört wird, wird der Lock im Destruktor automatisch freigegeben

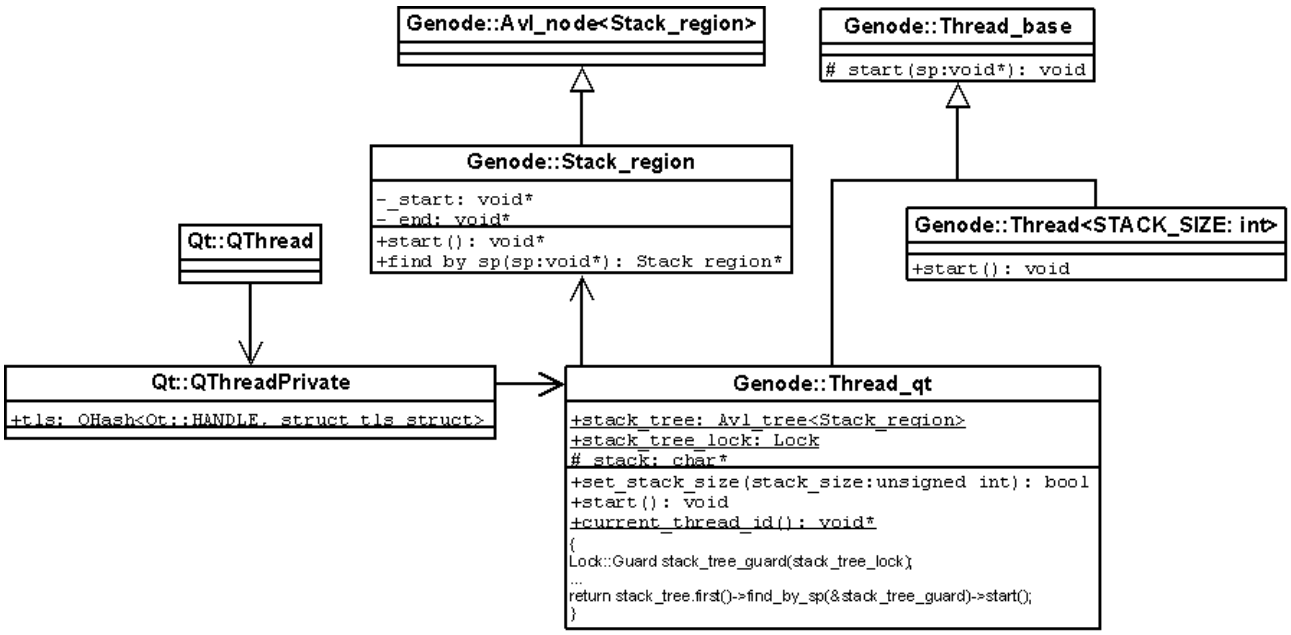

Abbildung 14: Thread-Implementierung

4.5 Portierung der QtGui-Bibliothek

4.5.1 Grafische Ausgabe

Die grafische Ausgabe von Qt erfolgt über den Nitpicker GUI-Server. Dieser ermöglicht die gemeinsame, überlappende Darstellung der grafischen Ausgaben mehrerer verschiedener Anwendungen auf dem selben physischen Ausgabegerät unter Wahrung spezieller Sicherheitskriterien. So soll z.B. garantiert werden, dass eine Anwendung nicht die Ausgaben anderer Anwendungen ausspionieren kann. Dieses Ziel wird dadurch erreicht, dass die einzelnen Anwendungs-Clients keinen Zugriff auf einen gemeinsamen Framebuffer erhalten, sondern stattdessen jeder Client die Möglichkeit bekommt, einen oder mehrere dedizierte Ausgabepuffer vom Nitpicker-Server anzufordern, die nur ihm selbst und dem Nitpicker-Server bekannt sind. Ein solcher Puffer kann dann vom Client z.B. als virtueller Framebuffer für die gesamte grafische Ausgabe der Anwendung verwendet werden, es ist aber ebenso möglich, jeweils einen separaten Puffer als Ausgabepuffer für einzelne Grafikobjekte (z.B. Fenster) zu verwenden. Damit der Pufferinhalt schließlich vom Nitpicker-Server auf dem Ausgabegerät dargestellt wird, müssen noch ein oder mehrere Ausgabebereiche („Views“) auf dem Puffer definiert werden. Neben der Position und Größe des darzustellenden Bereiches innerhalb des Grafikpuffers (Viewport) gehören zu den Eigenschaften eines Views auch die gewünschte Position auf dem Ausgabegerät sowie optional ein benachbartes View als Referenz für die Position des Views auf dem View-Stapel. Um die Grafikausgabe einer (Qt-)Anwendung über Nitpicker realisieren zu können, ist es also erforderlich, mindestens einen Ausgabepuffer vom Nitpicker-Server anzufordern und mindestens ein View darauf zu definieren und den Server anschließend über alle Änderungen des Pufferinhaltes und der Views zu informieren, damit dieser die Darstellung auf dem Ausgabegerät zum richtigen Zeitpunkt aktualisieren kann.

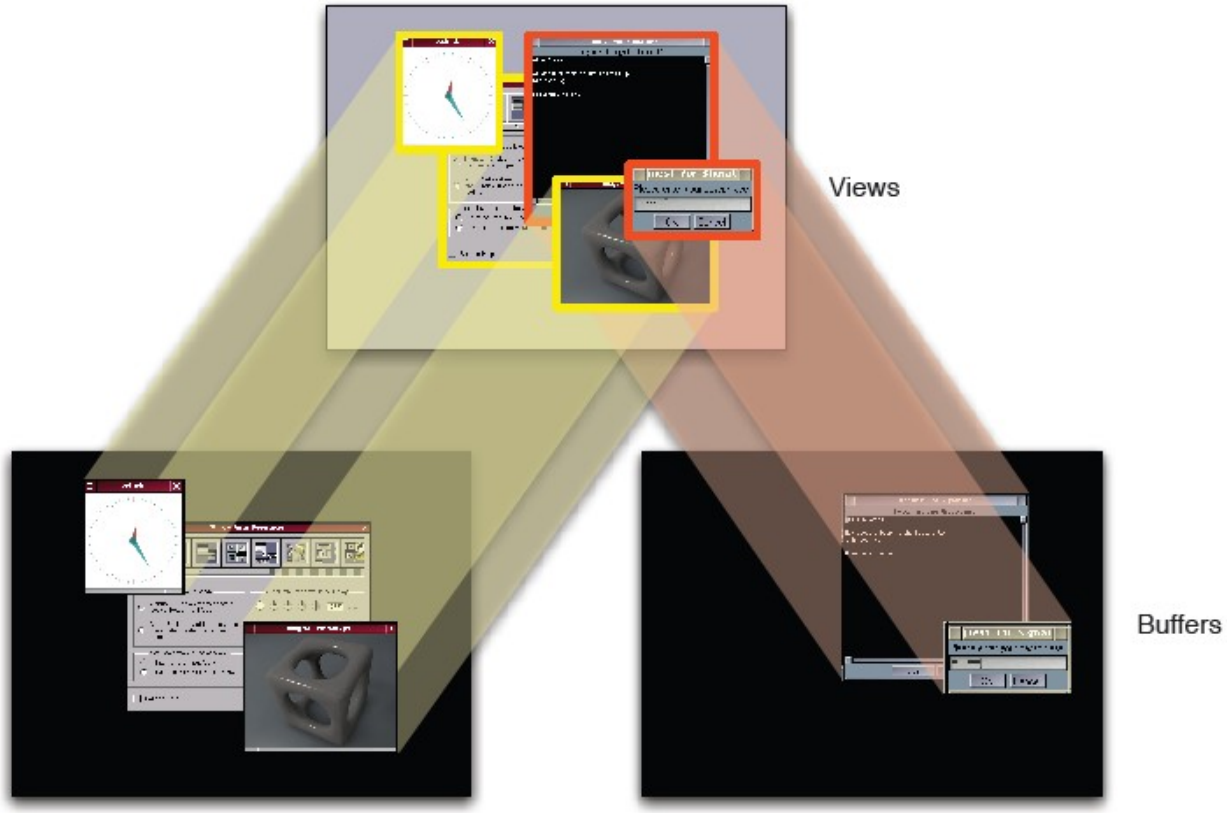

Abbildung 15: Nitpicker – Buffers und Views (Quelle: [4])

Qt unterstützt verschiedene Ausgabegeräte durch Treiberklassen, die von der Klasse `QScreen` abgeleitet sind. Zu den Aufgaben eines solchen Qt-Ausgabetreibers gehören die Initialisierung des Ausgabegerätes, die Ermittlung von Speicheradresse, Auflösung und Farbtiefe des Framebuffers, das Füllen des Framebuffers mit soliden Farbflächen, das Kopieren der Qt-Fensterinhalte in den Framebuffer sowie die Erzeugung der Grafikpuffer, die von den Fenstern zum Speichern des Fensterinhaltes verwendet werden.

Normalerweise teilen sich mehrere Qt-Anwendungen einen gemeinsamen Framebuffer ohne Zugriffsbeschränkungen. Es ist aber auch möglich, jede Anwendung als QWS-Server mit eigenem Framebuffer auszuführen und somit einzelnen Anwendungen dedizierte Ausgabegeräte zuzuordnen. Da ein gemeinsamer Framebufferzugriff den Sicherheitskriterien, die mit Nitpicker durchgesetzt werden sollen, entgegenlaufen würde und die Genode-Infrastruktur auch die zur Kommunikation zwischen den externen QWS-Clients und dem QWS-Server benötigten UNIX Domain Sockets nicht bereitstellt, wurde für die Genode-Portierung die letztgenannte Variante gewählt, d.h. jeder Qt-Anwendung sollte ein eigener, von anderen Qt-Anwendungen isolierter Nitpicker-Framebuffer zugeordnet werden. Da die Anzahl der vom Nitpicker-Server anforderbaren Framebuffer nur durch den verfügbaren Speicher des Clients begrenzt ist und die Abmessungen eines solchen Framebuffers auch nicht denen des physischen Ausgabegerätes entsprechen müssen, stand zusätzlich noch die Alternative im Raum, statt eines einzelnen Framebuffers in Vollbildgröße, der von allen Fenstern der selben Anwendung geteilt wird und auf dem für jedes Fenster ein View definiert wird, für jedes Fenster einen separaten Framebuffer mit den Abmessungen des Fensters anzufordern und darauf jeweils ein View zu definieren. Dies hätte bei Anwendungen mit wenigen, relativ kleinen Fenstern den Vorteil eines geringeren Speicherbedarfes, würde aber bei Fenstergrößenänderungen zu mehr Verarbeitungsaufwand führen, da in diesem Fall der Framebuffer jeweils neu allokiert werden müsste. Des Weiteren könnte bei dieser Implementierungsvariante der Gesamtspeicherbedarf der Anwendung im schlimmsten Falle (bei mehreren großen, sich überlappenden Fenstern) ein Vielfaches des Bedarfes der Ein-Framebuffer-Variante betragen, was die Reservierung einer dementsprechend höheren RAM-Quota erforderlich machen würde, die dann für andere Genode-Anwendungen nicht mehr zur Verfügung stünde. Aus diesen Gründen wurde diese Alternative nicht weiter verfolgt und die im Rahmen der Portierung neu erstellte Grafiktreiberklasse `QNitpickerScreen` arbeitet demzufolge mit einem einzigen Vollbild-Nitpicker-Framebuffer.

Für die Erzeugung und Aktualisierung der Views stellte die abstrakte Klasse `QWSWindowSurface` die geeignete Integrationsstelle dar. Sie repräsentiert ein Qt-Fenster aus Sicht des QWS-Servers und ist in gewisser Weise mit einem Nitpicker-View vergleichbar, da sie ebenfalls für die Darstellung eines gemeinsam mit einem Client genutzten rechteckigen Grafikbereiches zuständig ist und vom Client über alle relevanten Zustandsänderungen (betreffend Position, Größe, Stapelposition und Inhalt) dieses Bereiches informiert wird. Zu den konkreten Ausprägungen dieser abstrakten Klasse gehören insbesondere die Klassen `QWSLocalMemSurface` für Anwendungen, die als QWS-Server ausgeführt werden und somit mit einem gemeinsam nutzbaren Grafikpuffer im lokalen Adressraum auskommen sowie `QWSSharedMemSurface` für Anwendungen, die auf einen externen QWS-Server zugreifen und deshalb für einen gemeinsamen Grafikpuffer prozessübergreifendes Shared Memory benötigen. Weitere Varianten sind `QWSOnScreenSurface` für den Fall, dass die Zeichenoperationen eines Fensters ohne Zwischenpufferung direkt auf dem Framebuffer ausgeführt werden sollen sowie die Klasse `QWSDirectPainterSurface`, die als Basis für hardwarebeschleunigte Grafikausgabe dienen kann. Die Entscheidung darüber, welche der verschiedenen Subklassen für ein konkret zu erzeugendes Fenster instanziiert wird, wird dabei von der Grafiktreiberklasse in der Methode `createSurface()` getroffen. An dieser Stelle bot sich die Möglichkeit, statt einer Instanz der Klasse `QWSLocalMemSurface` eine Instanz einer von `QWSLocalMemSurface` abgeleiteten

neuen Klasse `QWSNitpickerWindowSurface` zu erzeugen, die sich gegenüber `QWSLocalMemSurface` dadurch auszeichnet, dass sie zusätzlich das benötigte Nitpicker-View erstellt und dieses bei jeder mitgeteilten Zustandsänderung des dazugehörigen Qt-Fensters ebenfalls auf den aktuellen Stand bringt. Entgegen den ersten Annahmen nach Betrachtung der Schnittstelle der `QWSWindowSurface`-Klasse stellte sich während der praktischen Umsetzung allerdings heraus, dass eine unmittelbare Aktualisierung des Nitpicker-Views in den gegebenen Methoden `setGeometry()`, `move()` und `flush()` nicht zu zufriedenstellenden Ergebnissen führt, da der Fensterinhalt teilweise erst nach Abschluss dieser Methodenaufrufe in den Framebuffer übertragen wird und deshalb der dargestellte View-Bereich zeitweise nicht mit der tatsächlichen Fensterdarstellung im Framebuffer übereinstimmte. Dieses Problem konnte jedoch durch eine zusätzliche Auswertung der Fensterereignis-Signale, die der QWS-Server nach Abschluss der Blit-Operationen emittiert, gelöst werden.

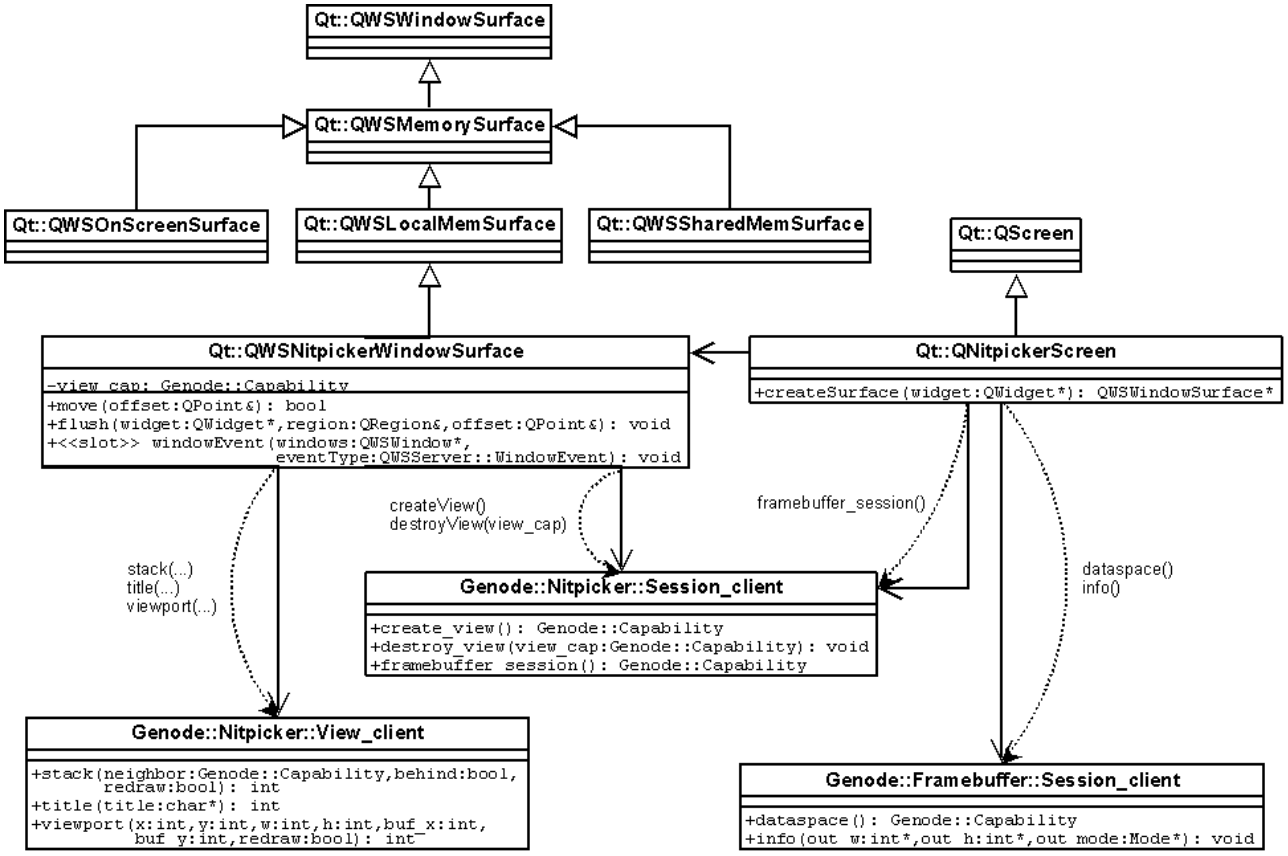

Abbildung 16: Nitpicker-Anbindung (Grafikausgabe)

4.5.2 Tastatur- und Mauseingabe

Die Eingabe von Tastatur- und Mausereignissen erfolgt ebenfalls über den Nitpicker GUI-Server. Dieser legt für jeden Client einen Ereignispuffer an, in welchem er für jedes Eingabeereignis, das innerhalb eines Views der Applikation stattfindet und unter der Voraussetzung, dass dieses View das gerade vom Nutzer fokussierte View ist, ein spezielles Ereignisobjekt ablegt, das anschließend vom Client ausgelesen werden kann. Ein solches Ereignisobjekt enthält generell für Tastatur- und Mausereignisse die Information, ob eine Taste gedrückt oder losgelassen wurde sowie einen Tastencode zur Identifizierung der jeweiligen Taste. Bei Tastaturereignissen entspricht dieser Tastencode dem Hardware-Scancode der Taste. Für Mausereignisse ist zusätzlich noch die Information enthalten, ob es sich bei dem Ereignis um die Bewegung des Mauszeigers oder des Mausrades handelt.

Qt unterstützt verschiedene Eingabegeräte durch Treiberklassen, die von den Klassen `QWSKeyboardHandler` bzw. `QWSMouseHandler` abgeleitet sind. Die Aufgabe dieser Treiberklassen ist es, die Eingabeereignisse des jeweils unterstützten Eingabegerätes aus der zugehörigen Datenquelle, üblicherweise einer Linux-Geräte-Datei, entgegenzunehmen und daraus die entsprechenden Qt-Ereignisse zu generieren. Da die Nitpicker-Eingabeereignisse alle im selben Ereignispuffer bereitgestellt werden und dieser nur sequenziell ausgelesen werden kann, war ein solcher unabhängiger Betrieb von Tastatur- und Maustreiber bei der Genode-Portierung allerdings nicht möglich. Um sicherzustellen, dass jeder der beiden Treiber nur die Eingabeereignisse verarbeitet, die auch für ihn bestimmt sind, wurde deshalb eine übergeordnete Klasse `QNitpickerInputHandler` erstellt, die die Ereignisobjekte aus dem Nitpicker-Eingabepuffer entnimmt und sie anschließend entsprechend ihrer Typkennung an den Tastatur- bzw. Maustreiber weitergibt. Der Tastaturtreiber selbst wurde von der vorhandenen Treiberklasse für Standard-PC-Tastaturen, `QWSPC101KeyboardHandler`, abgeleitet, da diese prinzipiell auch die von Nitpicker gelieferten Scancodes verarbeiten kann. Vor der Übergabe des Tastencodes an die Methode `QWSPC101KeyboardHandler::doKey()` muss lediglich noch der Tastenstatus, der von Nitpicker durch den Ereignistyp mitgeteilt wird, durch fallweises Setzen eines zusätzlichen Bits in den Tastencode hineincodiert werden. Die Maustreiberklasse wurde als Subklasse von `QWSMouseHandler` realisiert, welche über die Methode `mouseChanged()` die Koordinaten des Mauszeigers und den Status der Maustasten entgegennimmt. Diese Informationen können aus dem Nitpicker-Ereignisobjekt entnommen werden.

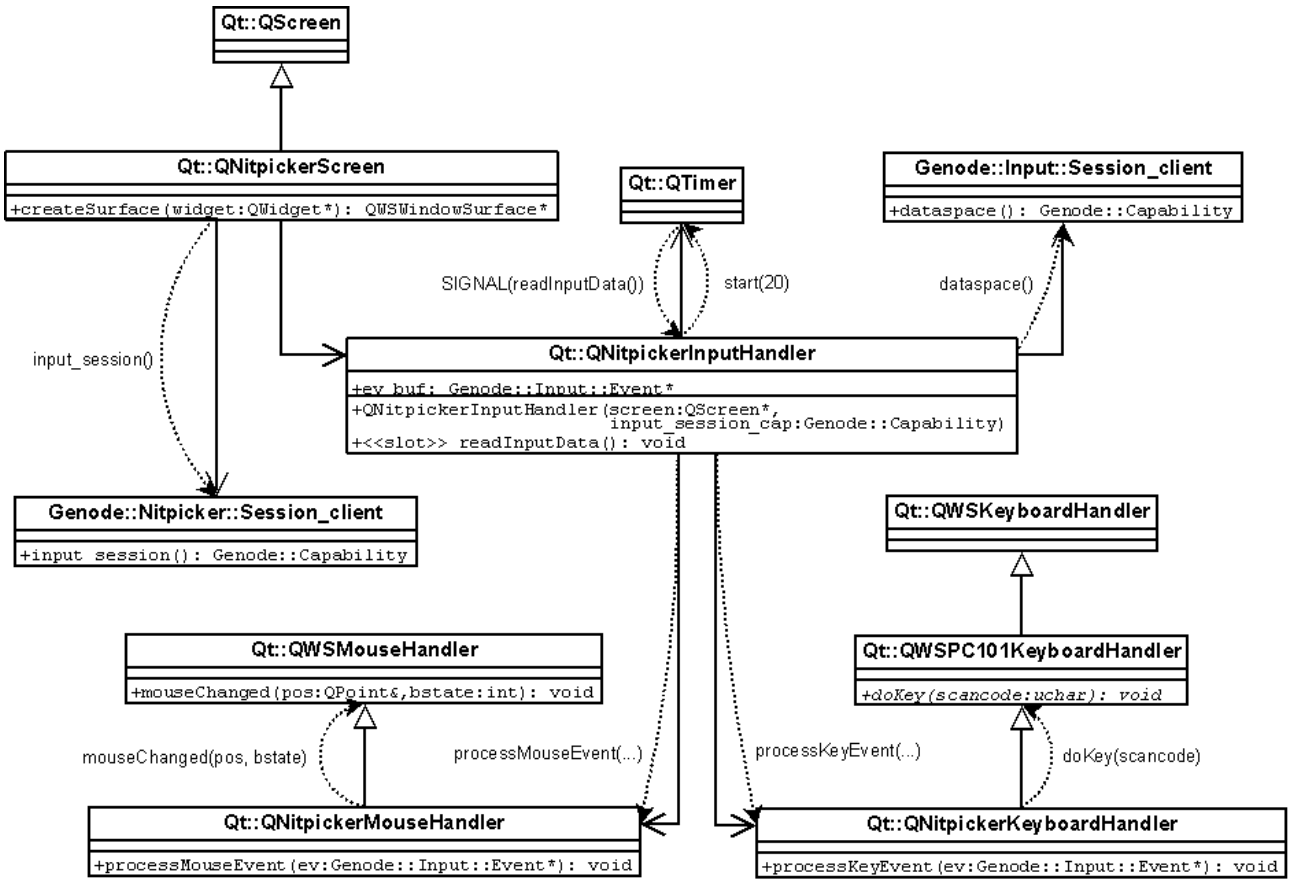

Abbildung 17: Nitpicker-Anbindung (Tastatur- und Mauseingabe)

5 Auswertung

5.1 Praxistauglichkeit

Ziel der Portierung war es, mit Qt neue grafische Anwendungen für Genode entwickeln zu können sowie bereits existierende Qt-Anwendungen auch auf Genode einsetzen zu können. Da die Entwicklung von Qt-Anwendungen für Genode derzeit nur auf einem externen (Linux-)Host-System möglich ist und bei der Portierung viel Wert auf die Beibehaltung der von Qt gewohnten plattformunabhängigen Anwendungsentwicklung gelegt wurde, kann die Praxistauglichkeit exemplarisch an Hand der Portierung einer bereits existierenden Beispielanwendung betrachtet werden.

Die zu portierende Anwendung sollte nach Möglichkeit eine sein, deren grafische Oberfläche mit dem Qt Designer erstellt wurde und die auch von dem Ressourcencompiler `rcc` und der Tastatur Gebrauch macht, um sich von der korrekten Funktionsweise dieser Qt-Features überzeugen zu können. Das Spiel Tetrix, das als Beispielanwendung mit dem Qt-Sourcecode mitgeliefert wird, kam diesen Anforderungen sehr nahe.

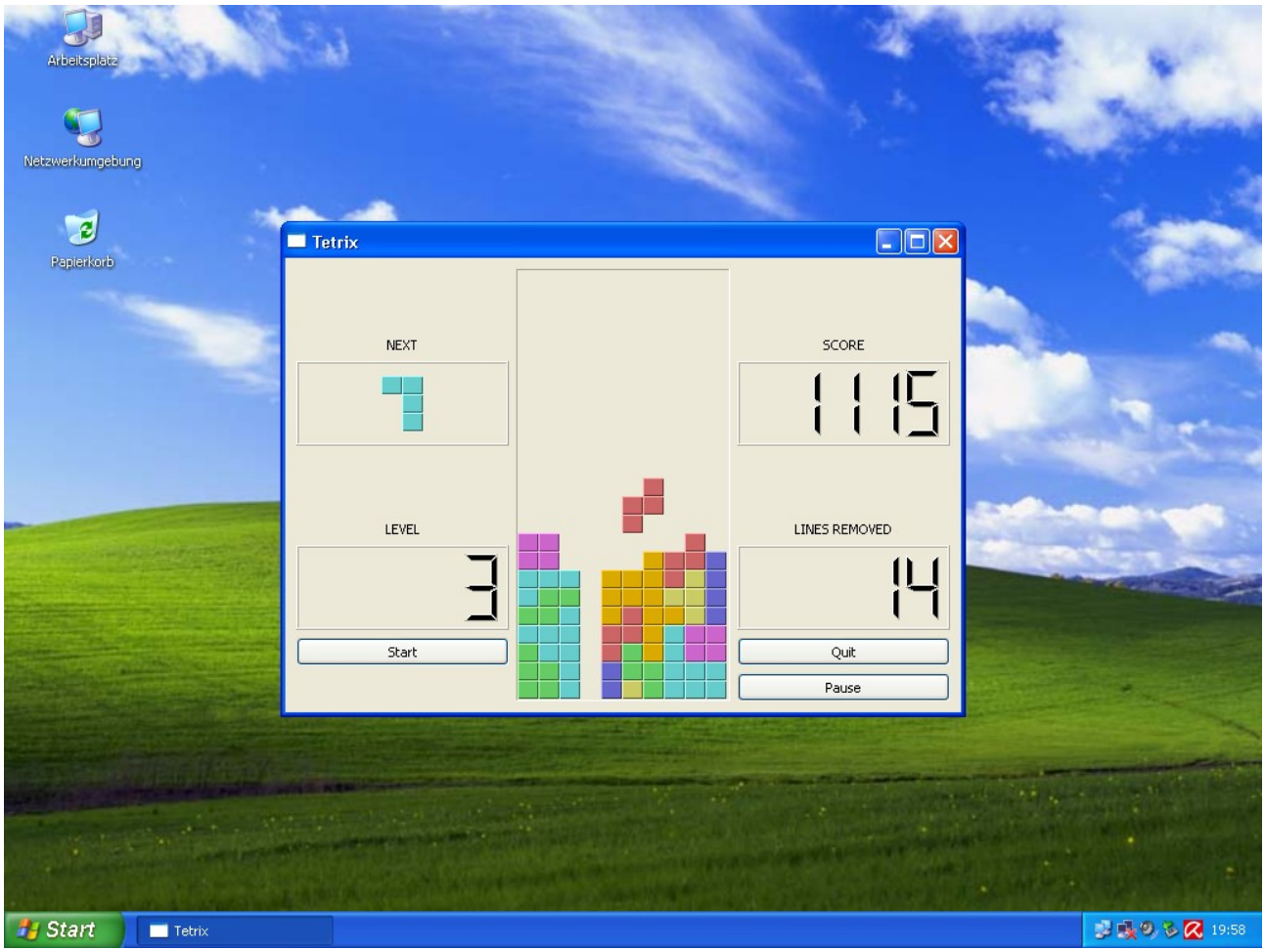

Abbildung 18: Tetrix auf Windows

Allerdings war diese Beispielanwendung von Trolltech eigentlich zur Demonstration der Scripting-Fähigkeiten von Qt konzipiert und benötigt deshalb zusätzlich zu den QtCore- und QtGui-Modulen noch die Qt-Module QtScript, QtXml und QtUiTools. Glücklicherweise stellte sich heraus, dass die Portierung dieser Module keinen allzu großen Aufwand darstellte, da diese größtenteils auf dem QtCore-Modul und vielen bereits portierten Funktionen aus der C-Bibliothek aufbauen. Für die

Portierung des QtXml-Moduls mussten lediglich die dazugehörigen Genode-Builddateien erstellt werden, für QtScript und QtUiTools mussten zusätzlich noch einige fehlende C-Funktionen in die `qt_c`-Bibliothek aufgenommen werden und im QtUiTools-Modul mussten noch die Code-Stellen auskommentiert werden, die für das Laden dynamischer Plugins zuständig sind, da dies mit Genode derzeit noch nicht möglich ist.

Nach der Portierung der zusätzlich benötigten Qt-Module und entsprechender Erweiterung des im Rahmen der Belegarbeit entwickelten Genode-Template-Projektes für Qt-Anwendungen war die Erzeugung einer lauffähigen Tetrix-Binärdatei im Genode-Buildsystem ohne jegliche Änderungen an Tetrix möglich. Dies war allerdings auch dem glücklichen Umstand geschuldet, dass Tetrix bereits so konfiguriert war, alle seine zur Laufzeit benötigten externen Dateien (dazu gehören die XML-Datei mit der Beschreibung der grafischen Oberfläche sowie die JavaScript-Dateien mit der Anwendungslogik) als Ressourcen in die ausführbare Datei mit einzubinden und zur Laufzeit aus dem Ressourcen-Dateisystem zu laden. Bei Anwendungen, die ihre externen Dateien standardmäßig aus dem POSIX-Dateisystem laden, wäre an dieser Stelle eine entsprechende Umsetzung auf das Ressourcen-Dateisystem notwendig, da auf Genode noch kein POSIX-Dateisystem vorhanden ist.

Die kompilierte Tetrix-Anwendung funktionierte größtenteils wie erwartet. Die Benutzeroberfläche wurde korrekt dargestellt und ohne merkliche Verzögerungen aufgebaut und aktualisiert. Die Tastatur- und Mauseingaben wurden ebenfalls korrekt und ohne merkliche Verzögerungen verarbeitet. Auffällig war allerdings, dass die Reihenfolge der Spielsteine bei jedem Neustart der Anwendung identisch war. Das liegt daran, dass der Zufallsgenerator-Algorithmus in der C-Support-Bibliothek mit der aktuellen Uhrzeit initialisiert wird, die aber unter Genode noch nicht ermittelt werden kann. Davon abgesehen war das Spiel jedoch sehr gut unter Genode spielbar.

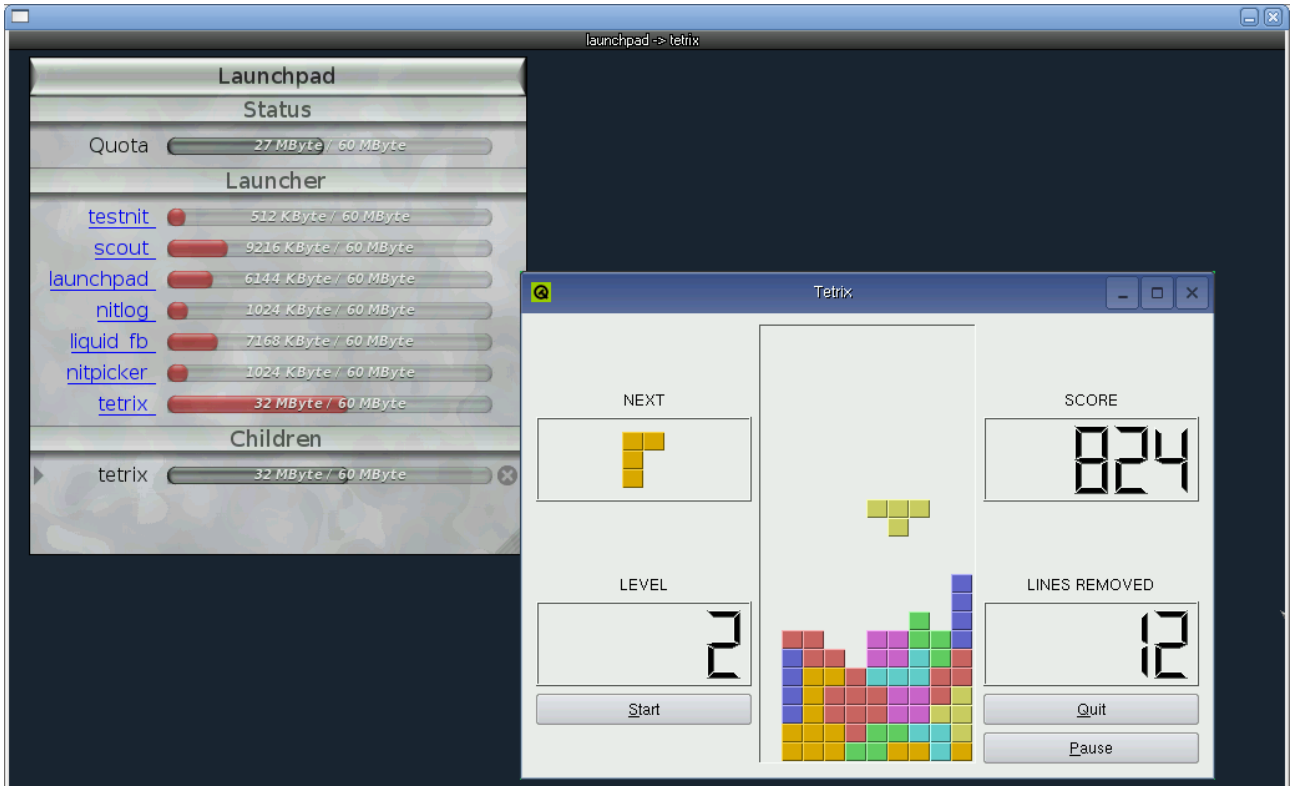

Abbildung 19: Tetrix auf Genode/Linux

5.2 Zukunftssicherheit

Im Folgenden sollen nun noch einige interne Details der Qt-Portierung untersucht werden. Interessant ist dabei vor allem die Zukunftssicherheit der Portierung im Hinblick auf die nächsten Qt-Versionen und im Hinblick auf andere, auch Capability-basierte Betriebssysteme.

Die Zukunftssicherheit der Portierung im Hinblick auf die nächsten Qt-Versionen hängt vor allem davon ab, wie viele Änderungen im Qt-Sourcecode vorgenommen wurden. Allgemein kann dazu gesagt werden, dass solche Änderungen nur dort vorgenommen wurden, wo es unbedingt nötig war (betrifft die Konfiguration der Zielplattform sowie die Grafik- und Eingabetreiberklassen) oder wo der Aufwand für die Implementierung der benötigten Funktionalität in der C-Support-Bibliothek zu hoch gewesen wäre (betrifft Timer, Threads und Synchronisationsmechanismen). Die vorgenommenen Änderungen bestehen dabei größtenteils aus Genode-abhängigem Code und zu einem kleinen Teil aus zusätzlich eingefügten Auswertungen eines Q_OS_GENODE Präprozessor-Makros, um beispielsweise nicht unterstützten Code wie z.B. das Laden dynamischer Plugins im QtUiTools-Modul zu deaktivieren.

Der Genode-abhängige Code in Qt umfasst ca. 751 Source Lines of Code (SLOC)[3], davon 323 SLOC im QtCore-Modul und 429 SLOC im QtGui-Modul. Das ist nicht sehr viel und lässt ein relativ problemloses Update auf zukünftige Qt-Versionen vermuten. Bei den während der Bearbeitungszeit der Belegarbeit durchgeführten schrittweisen Updates von ursprünglich „Qtopia Core“ 4.3.2 auf „Qt for Embedded Linux“ 4.4.1 war dies bis auf gelegentliche Patch-Anwendungs-Konflikte durch Abweichungen im Kontext auch der Fall.

Bei der Einschätzung des Portierungsaufwandes für zukünftige (Capability-basierte) Betriebssysteme ist neben dem Genode-spezifischen Code in Qt auch der Umfang der vorgenommenen Genode-Erweiterungen (Timed Semaphore und die spezielle Thread-Klasse) von Interesse. Hierfür konnten ca. 250 SLOC gemessen werden, was ebenfalls nicht sehr viel ist. Der Genode-abhängige Code in der C-Support-Bibliothek ist mit ca. 23 SLOC fast vernachlässigbar. Es kann also davon ausgegangen werden, dass der Portierungsaufwand für die im Rahmen dieser Belegarbeit portierte Qt-Funktionalität auf ein zukünftiges (Capability-basiertes) Betriebssystem nicht sehr hoch sein wird. Im Idealfall, d.h. wenn auf dem neuen System bereits eine POSIX C-Bibliothek vorhanden ist, müssten theoretisch nur die Builddateien und die Treiberklassen neu erstellt werden. Falls noch keine solche C-Bibliothek vorhanden ist, könnte wahrscheinlich ein großer Teil der für Qt auf Genode erstellten C-Support-Bibliothek auf dem neuen System wiederverwendet werden.

5.3 Optimierungsmöglichkeiten

Derzeit werden unter Genode nur statische Bibliotheken unterstützt. Durch den Einsatz von dynamischen Bibliotheken könnte der Speicherbedarf von Qt-Anwendungen noch verringert werden. Um eine genauere Aussage über das Einsparpotenzial treffen zu können, wäre es interessant zu wissen, wie hoch der Anteil der einzelnen Qt-Module ist, der in einer typischen Qt-Anwendung verwendet wird. Um dies herauszufinden, wurde mit dem `nm`-Programm jeweils eine Liste aller definierten Code- und Read-Only-Daten-Symbole im Tetrix-Binary und in den einzelnen Qt-Bibliotheken erstellt. Anschließend wurde mit Hilfe von `fgrep` jedes Symbol aus dem Tetrix-Binary einem Qt-Modul zugeordnet und daraufhin für jedes Modul die Summe der Symbolgrößen der in Tetrix gefundenen Symbole gebildet und der Summe aller in dem jeweiligen Modul definierten Symbole gegenüber gestellt.

3 gemessen mit `SLOCCount` von David A. Wheeler

Qt-Modul	Code und Read-Only-Daten in der Bibliothek (Genode)	Anteil in Tetrix (Genode)	Code und Read-Only-Daten in der Bibliothek (Linux)	Anteil in Tetrix (Linux)
QtCore	1.770.050 Byte	1.770.050 Byte (**100%**)	1.733.845 Byte	1.548.537 Byte (**89,3%**)
QtGui	5.399.840 Byte	5.396.223 Byte (**99,9%**)	5.481.292 Byte	5.064.993 Byte (**92,4%**)
QtScript	705.084 Byte	703.960 Byte (**99,8%**)	735.243 Byte	712.964 Byte (**97%**)
QtUiTools	360.073 Byte	359.829 Byte (**99,9%**)	359.847 Byte	359.639 Byte (**99,9%**)
QtXml	134.329 Byte	133.725 Byte (**99,6%**)	134.538 Byte	134.538 Byte (**100%**)

Tabelle 1: Anteil des Gesamt-Umfanges der einzelnen Qt-Module in Tetrix

Bei den Messergebnissen fällt auf, dass sich der Anteil der QtCore- und QtGui-Module zwischen der Genode-Version und der Linux-Version von Tetrix deutlich unterscheidet. In der Genode-Version scheint jedes Qt-Modul nahezu vollständig zur Anwendung gelinkt worden zu sein, während der Anteil in der Linux-Version teilweise geringer ausfällt, aber ebenfalls sehr hoch ist. Die Erklärung für den nahezu 100%-igen Anteil der Qt-Module in der Genode-Version besteht darin, dass die Bibliotheken unter Genode mit „`ld -r`“ erstellt werden und die daraus resultierenden monolithischen Objektdateien nur vollständig zur Anwendung gelinkt werden können, während unter Linux das `ar`-Programm zum Einsatz kommt, welches ein Bibliotheks-Archiv mit mehreren Objektdateien erstellt und dadurch einen feingranulareren Linkvorgang ermöglicht, der näher am tatsächlichen Bedarf der Anwendung liegt. Dass die Messung bei der Genode-Version nicht immer exakt 100% ergeben hat, liegt daran, dass einige Symbole mit dem GCC-spezifischen Attribut `weak` deklariert wurden und in mehreren Modulen vorkommen, aber nur einmal zur Anwendung gelinkt werden.

Der hohe Anteil der einzelnen Module unter Linux war allerdings auch ein wenig überraschend, besonders beim QtGui-Modul, denn eigentlich sieht die grafische Oberfläche von Tetrix doch eher schlicht aus und nicht so, als würde sie von der umfangreichen Funktionalität des QtGui-Modules großen Gebrauch machen. Hier lag die Vermutung nahe, dass der hohe Anteil des QtGui-Modules durch den Einsatz des QtUiTools-Modules bedingt war, welches die Generierung der Benutzeroberfläche aus einer zur Laufzeit geladenen XML-Beschreibung ermöglicht und dafür natürlich einen großen Umfang an GUI-Klassen für alle möglichen Fälle zur Verfügung haben muss. Um dieser Vermutung genauer nachzugehen, wurde eine zweite Messung an einer einfacheren Variante der Tetrix-Anwendung durchgeführt, die lediglich die QtCore- und QtGui-Module voraussetzt.

Qt-Modul	Code und Read-Only-Daten in der Bibliothek (Genode)	Anteil in Tetrix2 (Genode)	Code und Read-Only-Daten in der Bibliothek (Linux)	Anteil in Tetrix2 (Linux)
QtCore	1.770.050 Byte	1.770.0050 Byte (**100%**)	1.733.845 Byte	1.538.027 Byte (**88,7%**)
QtGui	5.399.840 Byte	5.396.223 Byte (**99,9%**)	5.481.292 Byte	4.757.894 Byte (**86,8%**)

Tabelle 2: Anteil des Gesamt-Qt-Umfanges der einzelnen Qt-Module in einer einfacheren Tetrix-Variante

Dabei ergab sich tatsächlich ein geringerer Anteil des QtGui-Modules unter Linux, der allerdings immer noch relativ hoch war. Dies lässt den Schluss zu, dass innerhalb der einzelnen Qt-Module wohl viele Abhängigkeiten zwischen den Klassen bestehen, so dass ein großer Teil des Bibliotheksumfanges in jeder Qt-Anwendung benötigt wird. Dadurch ergibt sich natürlich ein hohes Einsparpotenzial beim Einsatz dynamischer Bibliotheken - der Einsatz von dynamischen Bibliotheken würde sich bereits bei zwei Qt-Anwendungen lohnen.

5.4 Limitierungen und fehlende Funktionalität

Die Portierung ist, wie bereits erwähnt, noch nicht vollständig. Es fehlt noch die Unterstützung für ein POSIX-artiges Dateisystem, Netzwerkübertragungen, das Starten von neuen Prozessen, das Ermitteln von Datum und Uhrzeit, Soundausgabe sowie das dynamische Laden von Plugins. Drag-und-Drop sowie die Zwischenablagenfunktion sind derzeit nur innerhalb der selben Qt-Anwendung verfügbar, da die für einen anwendungsübergreifenden Einsatz dieser Features benötigte Nitpicker-Infrastruktur noch nicht implementiert ist. Weiterhin können Einschränkungen durch noch nicht implementierte Funktionen in der C-Support-Bibliothek auftreten.

5.5 Erfahrungen mit Genode

Die Erfahrungen mit Genode waren überwiegend positiv. Das Framework ist bereits sehr stabil und bietet einen Großteil der Funktionalität die von den QtCore- und QtGui-Modulen benötigt wird. Die wenigen während der Portierung aufgetretenen Bugs im Framework konnten von den Genode-Autoren erfolgreich behoben werden. Es gibt allerdings noch ein paar Limitierungen, wie z.B.:

- die unveränderliche Stackgröße des `main()`-Threads
- die limitierte Anzahl von Speicherregionen unter Genode/Linux
- die fehlende Möglichkeit, den Speicherbedarf eines Genode-Dienstes im Vorfeld erfragen zu können

welche größere Änderungen im Framework erfordern und erst zu einem späteren Zeitpunkt behoben sein werden. Das Problem der unveränderlichen Stackgröße des `main()`-Threads konnte mit Hilfe der neuen Thread-Klasse mit konfigurierbarer Stackgröße (siehe Kapitel 4.4.3) und einer Wrapper-`main()`-Funktion, die die eigentliche `main()`-Funktion in einem solchen Thread aufruft, zumindest umgangen werden. Eine Erweiterung des Frameworks um fehlende bzw. zusätzlich für Qt benötigte Funktionalität war durch das Repository-basierte Buildsystem auf unkomplizierte Weise möglich, dennoch galt es gerade hierbei, sich gründlich Gedanken über die Art und Weise der Implementierung zu machen, um die Grundprinzipien von Genode (minimale Komplexität der Komponenten und möglichst geringe Abhängigkeiten vom unterliegenden System) zu unterstützen.

6 Zusammenfassung und Ausblick

Das Ziel dieser Belegarbeit bestand darin, durch die Portierung des GUI-Toolkits Qt die Grundlage für die Entwicklung einer grafischen Benutzeroberfläche für Genode auf dem Fiasco-Mikrokern zu schaffen. Dieses Ziel wurde erreicht - durch die erfolgreiche Portierung der Qt-Module QtCore und QtGui ist nun eine komfortable Entwicklung neuer Anwendungen und die Portierung existierender Qt-Anwendungen auf Genode möglich. Durch den modularen Aufbau von Qt und die vollständige Isolierung der einzelnen Qt-Anwendungen mit Hilfe des Nitpicker GUI-Servers passt die Portierung auch gut in das allgemeine Sicherheitskonzept von Genode. Eine vollständige Portierung von Qt war zwar im Rahmen dieser Belegarbeit noch nicht möglich, diese wird aber im Anschluss an diese Arbeit von der Firma Genode Labs weiter vorangetrieben.

7 Literatur

[1] Genode-Übersicht: http://genode.org/documentation/general-overview
[2] Genode-News bei Heise Online: http://www.heise.de/newsticker/Genode-Sichere-Betriebssysteme-aus-Dresden--/meldung/114420
[3] Norman Feske, Christian Helmuth: Design of the Bastei OS Architecture. http://os.inf.tu-dresden.de/papers_ps/bastei_design.pdf
[4] Norman Feske, Christian Helmuth: A Nitpicker's guide to a minimal-complexity secure GUI. http://os.inf.tu-dresden.de/papers_ps/feske-nitpicker.pdf
[5] Genode Labs: http://www.genode-labs.com
[6] moc-Dokumentation: http://doc.trolltech.com/4.4/moc.html
[7] rcc-Dokumentation: http://doc.trolltech.com/4.4/rcc.html
[8] uic-Dokumentation: http://doc.trolltech.com/4.4/uic.html
[9] qmake-Dokumentation: http://doc.trolltech.com/4.4/qmake-manual.html
[10] Qt Designer: http://doc.trolltech.com/4.4/designer-manual.html
[11] Qt Linguist: http://doc.trolltech.com/4.4/linguist-manual.html
[12] Qt Assistant: http://doc.trolltech.com/4.4/assistant-manual.html
[13] Qt Embedded Whitepaper: http://trolltech.com/pdf/Qt%20Embedded/qt-embedded-43-whitepaper-a4.pdf
[14] Carsten Weinhold: Portierung von Qt auf DROPS. http://os.inf.tu-dresden.de/papers_ps/weinhold-beleg.pdf
[15] Norman Feske, Hermann Härtig: Demonstration of DoPE - a Window Server for Real-time and Embedded Systems. http://www.genode-labs.com/veroeffentlichungen/dope-rtss-2003.pdf
[16] Scout widget set: http://www.genode-labs.com/products/graphical-user-interfaces
[17] Xynth Whitepaper: http://alperakcan.org/projects/xynth/down/xynth-doc-whitepaper.pdf
[18] SDL: http://www.libsdl.org
[19] GTK+: http://www.gtk.org
[20] Trolltech Dokumentation: http://doc.trolltech.com