Entwicklung einer grafischen Benutzeroberfläche für Brettspiele mit Qt

Eine Projektarbeit von

Philipp Meißner

Inhalt

Einleitung

Ziel dieses Projektes ist es, eine grafische Anwendung zu erstellen mit der man das gälische Strategiespiel UISGE spielen kann. Hierfür soll die Qt-Klassenbibliothek genutzt werden. Es handelt sich dabei um ein freies Framework, mit dem man unter anderem ansprechende grafische Benutzeroberflächen erschaffen kann. Qt funktioniert mit verschiedenen Betriebssystemen und bietet neben der Erstellung von Benutzeroberflächen noch etliche andere Funktionen, von denen einige im Laufe dieser Arbeit beschrieben werden.

In der vorliegenden Dokumentation werden häufig Bezeichner aus dem erstellten Programmcode genannt. Ist ein Wort ein unveränderter Bezeichner, der genauso im Quellcode steht, so ist es durch eine graue Farbe und Kursivschrift gekennzeichnet - zum Beispiel *QGraphicsScene* oder *deCodeROW*.

Clean Code

Der Stil, in dem ein Computerprogramm geschrieben ist, ist ein entscheidender Qualitätsfaktor. Er bestimmt im Wesentlichen die Lesbarkeit und somit auch die Wartbarkeit des Codes. Eines der berühmtesten Bücher über Programmierstil ist das Buch „Clean Code“, in dem sich der Software-Experte Robert C. Martin umfassend mit dieser Thematik auseinandersetzt. Der Stil, der in diesem Buch empfohlen wird, soll auch diesem Projekt als Richtlinie dienen, wobei es auch andere Einflussfaktoren gibt. So wird zum Beispiel aus Konsistenzgründen teilweise der Stil des Qt-Frameworks kopiert, da es sich ja bei dem zu entwickelnden Brettspiel-Framework um eine Art Bibliothek handelt, die zusammen mit dem Qt-Framework benutzt wird und dies dem Anwender ein ständiges Umdenken erspart. Jedoch gilt das Qt-Framework als eines der elegantesten Softwareprojekte der Welt und sein Stil beißt sich keineswegs mit den Empfehlungen des Buches „Clean Code“. Konkret wird der Stil der Setter- und Getter-Funktionen übernommen. Die Getter-Funktionen haben dabei den Namen der Variablen und das Schlüsselwort „const“, die Setterfunktionen haben das Präfix „set“. Beide Funktionen sind aus Geschwindigkeitsgründen inline und müssen daher auch in der Headerdatei deklariert werden. Sonstige Funktionen stehen natürlich nicht in der Headerdatei.

Listing 1: Setter- und Getter-Funktion im Stil des Qt-Frameworks

```
//in Datei Example.h
class Example
{
public:
        Example(); //Konstruktor
        void setVar();
        int Var() const;
private:
        int m_Var;
}
inline int Example::Var() const {return m_Var;}
inline void Example::setVar(int Var) {m_Var = Var;}
```

Aus dem Buch „Clean Code“ wurden unter anderem die Empfehlungen für die vertikale Formatierung übernommen. Danach sollte eine Quelltextdatei typischerweise ca. 200 Zeilen lang sein und eine Länge von 500 Zeilen nicht überschreiten. Funktionen sollen so kurz wie möglich sein. Sie sollten im Regelfall wenige Zeilen umfassen und nur in Ausnahmen 20 Zeilen überschreiten. Sehr kritisch behandelt der Autor Kommentare. Da diese vom Compiler nicht beachtet werden, besteht stets die Gefahr, dass sie nach einer Änderung des Programmcodes, bei dem versäumt wurde, die Kommentare anzupassen, nicht mehr aktuell sind und falsche Informationen enthalten. Daher sollte der Fokus weniger auf eine ausführliche Kommentierung als auf einen selbsterklärenden Programmcode gelegt werden. Neben der bereits erwähnten Bestrebung hin zu kurzen Funktionen ist daher auch die Verwendung von selbsterklärenden und aussprechbaren Bezeichnern wichtig. Eine einzeilige Funktion mit dem Namen „setHeight“, die eine Variable namens „NewHeight“ als Parameter übernimmt, sollte keinen Kommentar benötigen, der erklärt, was die Funktion tut. Ist dies dennoch nötig, ist der Name ungünstig gewählt. Kommentare sollten nur dort eingesetzt werden, wo sie zum Erklären eines Konzepts wirklich nötig sind. Jedes Mal, wenn ein Kommentar benötigt wird, weil der Programmcode nicht selbsterklärend genug ist, sollte auch die Ursache dafür hinterfragt werden. Grundlage all dieser Bestrebungen ist die Erkenntnis, dass ein Programmierer ungefähr zehnmal mehr Zeit damit verbringt, bereits geschriebenen Code zu lesen als neuen zu schreiben. Das Lesen eines Codes zu erleichtern, erleichtert daher auch das Schreiben neuen Codes ungemein.

Projektplanung

Um eine starke Entkopplung zwischen der Spiellogik und der Darstellung sicherzustellen, soll die Entwicklung bewusst in zwei Schritten ablaufen. Zunächst soll ein allgemeines Framework zum Erstellen grafischer Darstellung von Brettspielen erstellt werden und anschließend soll ein Stück Software entwickelt werden, das dieses Framework nutzt, um aus dem bereits vorhandenen Uisge-Code eine allein lauffähige grafische Anwendung zu machen. Der bereits existierende Code soll dabei möglichst beibehalten werden, jedoch wird er unter Umständen stellenweise erweitert, um alle Funktionen des Frameworks nutzen zu können.

Das Framework soll nicht speziell auf das Spiel Uisge zugeschnitten, sondern allgemein für die Darstellung von Brettspielen geeignet sein, bei denen Spielfiguren auf Feldern bewegt werden, die in Reihen und Spalten angeordnet sind. Neben Uisge könnte man somit auch Schach, Dame oder Go mit dem Framework realisieren. Abstrahiert man die Regeln derartiger Spiel,e kommt man auf folgende Gemeinsamkeiten:

- Es gibt ein Spielfeld, bei dem Felder in Zeilen und Spalten angeordnet sind, wobei die Felder nicht zwingend quadratisch sein müssen, auch wenn dies bei den bekannten Fällen so ist.

- Es gibt Spielfiguren. Diese Spielfiguren können sich bewegen, neu hinzukommen, verschwinden oder sich in andere Figuren umwandeln.

Aus diesen Eigenschaften ergeben sich die Anforderungen an das Framework. Unter Umständen könnte man die Darstellung eines gerasterten Spielfeldes auch zum Realisieren eines Minesweeper-Klons oder eines ähnlichen Spiels verwenden, jedoch werden diese bei der Entwicklung nicht explizit bedacht. Als Anregung für das Design soll die 2D-Ansicht des Spiels Chess Titans dienen, welches in Windows 7 und Vista enthalten ist. Es ist ein sehr schön gestaltetes Schachspiel, welches intuitiv zu bedienen ist und sich gut zur Verwendung mit einem Touchscreen eignet. Abbildung 1 zeigt eine Spielsituation in Chess Titans. Das Programm zeigt ein Spielfeld und die Spielfiguren an. Zusätzlich benutzt es verschiedene Akzente, um aktuell wichtige Felder und Figuren hervorzuheben. Eine weitere wichtige Darstellungsfunktion ist dabei das Skalieren. Bei Chess Titans verändert sich beim Vergrößern und Verkleinern des Programmfensters automatisch die Größe des Spielfelds und aller Figuren, so dass sie das Fenster optimal ausfüllen.

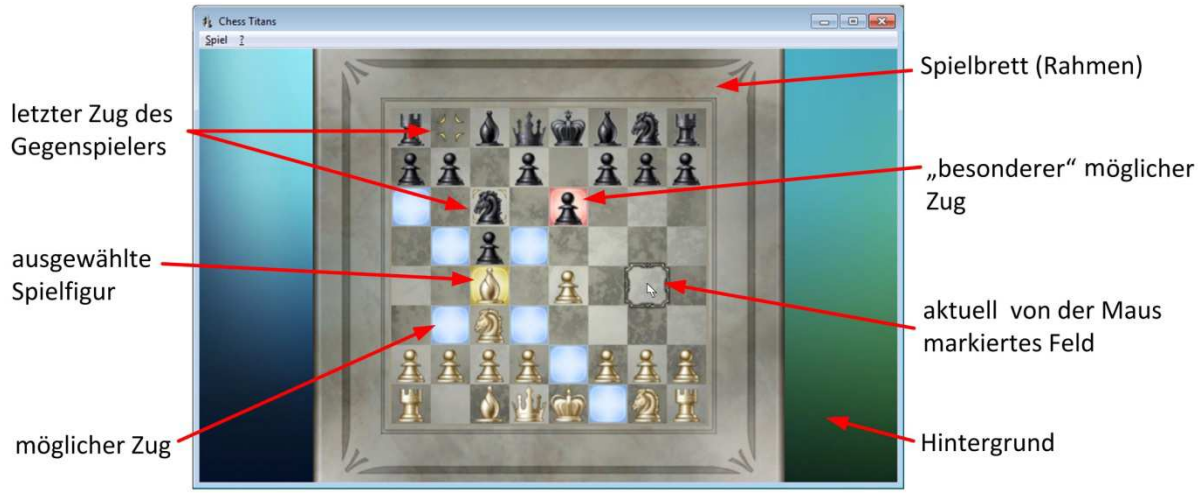

Abbildung 1: Chess Titans - Weiß am Zug

Bedenkt man nun zusätzlich zu den Spielregeln und der grafischen Darstellung auch die Tatsache, dass das Programm mit einer Maus oder einem Touchscreen bedient wird, erhält man den Anforderungskatalog an das Framework:

- Darstellung eines Spielfelds
- Darstellung der Spielfiguren
- flüssiges lineares Bewegen der Spielfiguren
- Darstellung von Akzenten und Markierungen
- Darstellung des Hintergrunds
- automatische Skalierung aller grafischen Elemente bei Veränderung der Fenstergröße
- Entgegennehmen von Mausklicks und deren Zuordnung zu den Spielfeldern

Nach der Programmierung des Frameworks wird dieses benutzt um aus dem vorhandenen Uisge-Programm eine grafische Anwendung zu machen. Der Code soll dabei möglichst wenig verändert werden. Das Aussehen des Spiels soll schlicht sein und an den keltischen Ursprung erinnern. Es soll ein einfaches Kästchenraster werden, das mit einem keltischen Knotenmuster umrandet ist. Kleine zusätzliche Akzente, wie eine Holz- oder Lederstruktur der Spielfläche sind zusätzlich denkbar.

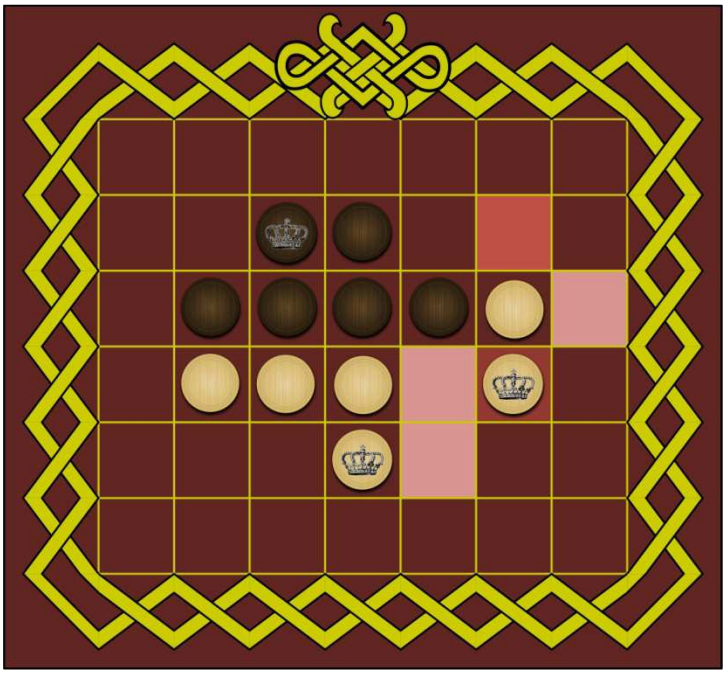

Abbildung 2: Grobentwurf des Spieldesigns

Das Brettspiel-Framework

Das Brettspiel-Framework besteht aus drei Klassen, die zusammen in der Lage sind, die meisten denkbaren Aspekte der grafischen Darstellung eines Brettspiels abzudecken. Mit ihnen wird nicht nur die Darstellung des Spielfeldes und der Spielfiguren realisiert, sondern auch eine lineare Bewegung der Spielsteine ermöglicht. Besonders wichtig ist dabei, dass alle grafischen Objekte stufenlos und flüssig skaliert werden, damit die Zeichenfläche, welche durch die Größe des Programmfensters vorgegeben ist, stets optimal ausgenutzt wird. Ebenso muss das Framework Mausklicks verarbeiten und sie den einzelnen Spielfeldern zuordnen. Da das Framework zur Erfüllung seiner Aufgabe die Position der grafischen Repräsentationen der Spielsteine verwalten muss, soll es sinnvollerweise auch die gesamte Spielsteinverwaltung übernehmen können.

Die Vorbereitung

Um in einem Qt-Programm Grafiken darstellen zu können, benötigt man ein Objekt des Typs *QGrapicsView*, das man einfach im Formulareditor in das Programmfenster einfügen kann. In dem Editor kann man dem Objekt auch einen Namen zuweisen, über den man dann später im Programm darauf zugreifen kann. Ein *QGraphicsView*-Objekt ist ein Fenster auf eine so genannte Scene, welche wiederum ein Objekt des Typs *QGraphicsScene* ist. Eine Scene ist eine Art Leinwand, auf der man verschiedene Grafikobjekte platzieren kann. Das Objekt *QGrapicsView* – auch einfach nur View genannt – zeigt dann einen Ausschnitt der ihm zugeordneten Scene.

Um also Grafikobjekte anzeigen zu können, muss zunächst im Formulareditor ein View erstellt werden. Anschließend muss im Programm eine neue Scene erstellt und dem View zugewiesen werden. Dies ist in Listing 2 gezeigt, wobei der Name des Views vorher im Formulareditor auf „MainView“ festgelegt wurde.

Listing 2: Initialisieren eines Views

```
QGraphicsScene *MainScene = new QGraphicsScene(this); //Erstellen einer neuen Scene
ui->MainView->setScene(MainScene); //Zuweisen der Scene zum im Formulareditor erstellten View
ui->MainView->setAlignment(Qt::AlignLeft | Qt::AlignTop); //Ausrichten: oben links = Pos 0/0
```

Im Formulareditor wurde zudem das Aussehen des Programms so konfiguriert, dass der View stets die gesamte Fensterfläche unterhalb der Menüleiste einnimmt. Somit war ein Grundgerüst geschaffen und es konnte mit der Entwicklung des Frameworks begonnen werden.

Das Spielbrett

Das Spielbrett ist die Hauptkomponente des Frameworks, es übernimmt die Darstellung des Spielbrettbildes und kann Positionen auf der Grafik den einzelnen Spielfeldern zuordnen. Es kümmert sich auch darum, dass die Grafik stets optimal skaliert und im Programmfenster zentriert ist. Es ist in der Klasse „*Board*“ implementiert, deren cpp-Datei knapp 200 Zeilen lang ist. Damit die Klasse *Board* ein Bild anzeigen kann, erbt sie von der Klasse *QGraphicsPixmapItem*, welche ein Grafikobjekt zur Darstellung in einer Scene repräsentiert. Außerdem wird sie durch das Erben von der Klasse *QObject* und dem Einfügen des Ausdrucks „Q_OBJECT“ am Anfang der Deklaration zu einem *QObject* und kann somit das Qt-Eventsystem nutzen. Hier gibt es einen kleinen Fallstrick. Wenn man eine Klasse nicht beim Erstellen durch eine entsprechende Auswahl im Neue-Datei-Dialog zu einem QObject macht, sondern dies nachträglich durch das Einfügen der entsprechenden Schlüsselwörter in den Quelltext realisiert, sollte man anschließend qmake ausführen, da es sonst beim Erstellen des

Programms zu einer Fehlermeldung kommen kann. Man findet den Befehl dafür im Erstellen-Menü des Qt Creator.

Beim Erstellen wird dem Spielbrett-Objekt ein Zeiger auf den View übergeben, welchen es nutzt, um sich in seinem Konstruktor selbst der Scene, welche mit dem View verbunden ist, hinzuzufügen. Der Zeiger wird außerdem auch dafür gebraucht, um die aktuelle Größe des Views abfragen und das Spielfeld entsprechend skalieren zu können. Für die Initialisierung des Spielfelds müssen zwei Memberfunktionen des Objekts aufgerufen werden. Die Funktion *setImage* hat einen String als Parameter, mit dem Speicherort und Name einer Bilddatei übergeben werden können. In der Funktion wird daraus ein Objekt des Typs *QImage* erstellt, aus dem dann ein Objekt des Typs *QPixmap* generiert wird. Diese Pixmap ist Bestandteil des Objekts *QGraphicsPixmapItem* und wird von diesem auf der ihm zugeordneten Scene zur Darstellung gebracht. Bei der Generierung der Pixmap können verschiedene Optionen ausgewählt werden. So gibt es einen Befehl, mit dem eine Pixmap auf eine angegebene Größe skaliert wird, wobei eingestellt werden kann, ob die Grafik dabei verzerrt werden darf oder ihr Seitenverhältnis beibehalten muss. Diese Methode wurde für sämtliche Skalierfunktionen des Frameworks verwendet.

Der zweite Teil der Spielbrettinitialisierung erfolgt durch den Aufruf der Methode *DefinePlayingField*. Die Parameter dieser Methode sind zum einen ein Objekt des Typs *QRect*, das ein Rechteck repräsentiert, welches beschreibt, in welchem Bildbereich der Spielbrettgrafik sich das eigentliche Spielfeld befindet, sowie zwei Integer-Variablen, die die Anzahl der Reihen und Spalten des Spielfelds angeben. Die Angaben zu Größe und Position des Spielfeldbereichs erfolgen dabei in Pixeln relativ zur oberen linken Ecke der Spielbrettgrafik, wobei die Gesamtgröße der Grafik einfach aus dem vorher erzeugten *QImage* Objekt ausgelesen werden kann. Die Funktion erstellt zudem eine Anzahl Objekte des Typs *QGraphicsLineItem*, die in einer Liste, die Teil des Spielbrettobjekts ist, verwaltet werden. Diese Objekte sind Linien, die als Trennstriche zwischen den Spielfeldern angezeigt werden können. Das Spielbrettobjekt beinhaltet auch ein Objekt des Typs *QGraphicsRectItem*, welches zum Darstellen einer Spielfeldumrandung genutzt werden kann. Das Aussehen der Trennlinien und der Spielfeldumrandung kann über eine Funktion verändert werden und mit zwei weiteren Funktionen kann deren Sichtbarkeit unabhängig voneinander aktiviert bzw. deaktiviert werden. Abbildung 3 zeigt ein Spielbrett mit deutlichen roten Trennlinien und Rahmen.

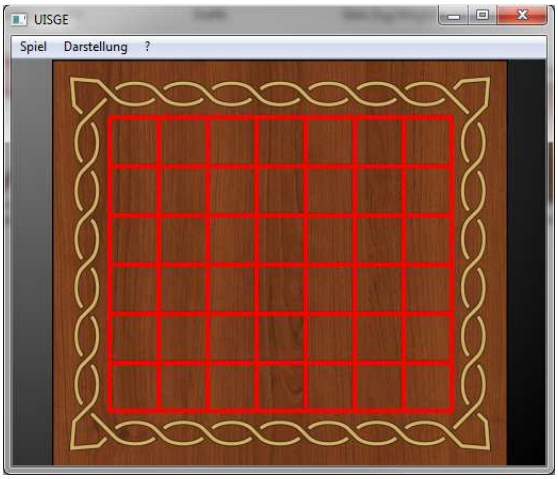

Abbildung 3: Rahmen und Trennlinien

Die wichtigsten Aufgaben des Spielbrettobjekts sind das Skalieren und Positionieren der Spielbrettgrafik sowie die Verarbeitung von Mausinteraktionen. Dafür muss es in der Lage sein, eine Veränderung der Programmfenstergröße oder eine Mausbewegung zu erkennen. Dies wird durch die Verwendung des Qt-internen Eventsystems erreicht.

Jedes Mal, wenn die Größe des Programmfensters verändert wird, wird vom Qt-Framework die Methode *resizeEvent* des Hauptfensterobjekts aufgerufen. Es handelt sich dabei um eine virtuelle Methode, die überladen werden kann, um ein Individuelles Reagieren auf dieses Ereignis zu implementieren. Das Vorgehen ist in, Listing 3 gezeigt.

Listing 3: Realisierung eines Event-Handles für den resizeEvent

```
//Hauptfenster-Klassendeklaration in der Datei "mainwindow.h"
public:
  void resizeEvent(QResizeEvent * event);


//Implementierung der Methode in der Datei mainwindow.cpp
void MainWindow::resizeEvent(QResizeEvent * event)
{
  //Hier kann individuell auf eine Änderung der Fenstergröße reagiert werden.
  //Das übergebene Event-Objekt enthält die neue und die alte Fenstergröße.
}
```

In dem Eventhandler wird die Methode *RescaleFast* des Spielbrettobjekts aufgerufen. In dieser wird die Größe des Views ausgewertet und die Pixmap, welche die Spielfeldgrafik enthält, entsprechend skaliert. Anschließend wird sie in der Methode *PlaceInView* im Viewfenster zentriert und die Methode *DrawGrid* aufgerufen, die den Spielfeldrahmen und die Trennlinien neu positioniert.

Es hat sich gezeigt, dass das erzeugen einer neuen, korrekt skalierten Pixmap aus dem *QImage* Objekt so zeitintensiv ist, dass es merklich ruckelt, wenn das Fenster durch Ziehen mit der Maus vergrößert wurde und sich somit die Fenstergröße sehr schnell hintereinander geändert hat. Um dies zu vermeiden und ein flüssiges Ändern der Fenstergröße zur ermöglichen, wurde ein System implementiert, dass zwei verschiedene Skalierungsvarianten nutzt. In der Funktion *RescaleFast* wird nicht die Bilddatei des Spielfelds, welche Teil des *QImage* Objekts ist, als Grundlage für die neue Pixmap genommen, sondern die aktuell verwendete Pixmap. Dies geht wesentlich schneller, hat jedoch den Nachteil, dass es zu merklichen Qualitätseinbußen kommt. Wenn eine kleine Pixmap als Grundlage einer größeren dient, wird die größere unscharf. Der Effekt verschlimmert sich, wenn die Fenstergröße sehr häufig verändert und dabei stets die alte Grafik zum Erzeugen der neuen herangezogen wird. Abbildung 4 zeigt ein Spielbrett, nachdem das Fenster mit der Maus mehrmals groß und wieder klein gezogen wurde, und Abbildung 5 zeigt dasselbe Spiel nach einer Neuberechnung auf Basis des *QImage* Objekts.

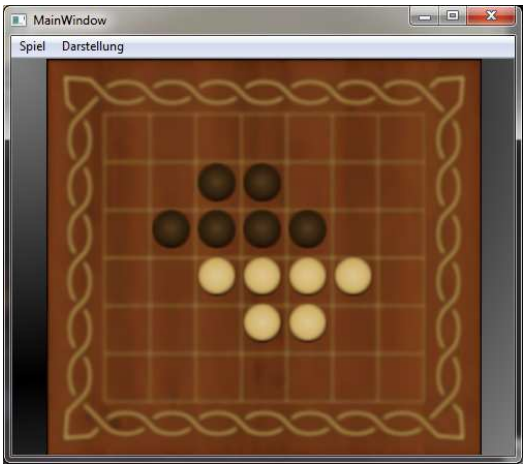

Abbildung 4: Spielfeld nach dem schnellen Skalieren

Abbildung 5: Spielfeld nach dem sauberen Skalieren

Da also nur das Erzeugen der skalierten Pixmap aus dem *QImage* Objekt zu einer schönen Darstellung führt, jedoch nur das Skalieren mit Bezug auf die alte Pixmap schnell genug für eine flüssige Bewegung ist, wurden beide Varianten miteinander verknüpft. Wie bereits erwähnt wird in der Funktion *RescaleFast*, die vom *rescaleEvent* ausgelöst wird, die schnelle, unsaubere Variante verwendet. Zusätzlich wird jedoch ein Timer gestartet, der, wenn er abgelaufen ist, ein Neuzeichnen des Spielfelds mit der langsamen, aber sauberen Methode auslöst. Bei jedem neuen Aufruf der *RescaleFast* Funktion wird der Timer zurückgesetzt und läuft somit niemals aus, solange die Funktion regelmäßig aufgerufen wird. Erst wenn der Benutzer mit dem Skalieren der Fenstergröße fertig ist, wird die Funktion nicht mehr aufgerufen. Der Timer läuft ab und erzeugt ein Event, das die Methode *RescaleClean* startet, in der die Pixmap aus dem *QImage* Objekt neu erstellt wird. Die Laufzeit des Timers ist standardmäßig auf 300 Millisekunden eingestellt, kann aber mit der Methode *setReloadDelayInMS* verändert werden. Die Methode *RescaleClean* löst zudem einen eigenen Event aus, den andere Objekte empfangen können, um z. B. selbst ein sauberes Neuzeichnen ihrer Grafik auszuführen. Das ganze Verfahren nutzt also nicht nur die systemeigenen Events, wie den *resizeEvent*, sondern verwendet das Eventsystem auch, um Nachrichten zwischen eigenen Objekten auszutauschen und einen definierten zeitlichen Ablauf zu implementieren. Das ganze Vorgehen ist in Listing 4 zu sehen, wobei zur leichteren Verständlichkeit die Methoden zum Teil gekürzt aufgeführt sind.

Listing 4: Verwendung des Eventsystemsystems

```
//Deklaration der Klasse Board (nur ausschnittweise)
class Board : public QObject, private QGraphicsPixmapItem
{
  Q_OBJECT //Wichtig, damit das Eventsystem genutzt werden kann
public:
  void RescaleFast(); //wird in rescaleEvent des Hauptfensters aufgerufen

signals: //Eigene Events
  void CleanRedrawDone();

private slots: //Empfänger von Events
  void RescaleClean();

private:
  QTimer ImageReloadTimer; //ein Timer-Objekt (löst selbst Events aus)
  int ReloadDelay;
  QImage BoardImage;  //hier wird Spielbrettgrafik gespeichert
}



//Konstruktor der Klasse Board (vereinfacht)
Board::Board()
{
  ImageReloadTimer.setSingleShot(true); //Timer löst ein Event aus und hält dann an.
  ReloadDelay = 300;

  Verbinden des Timeoutsignals des Timers mit dem Slot (Eventempfänger) RescaleClean
  connect(&ImageReloadTimer,SIGNAL(timeout()),this,SLOT(RescaleClean()));
}



//Diese Methode macht das schnelle, unsaubere Skalieren. Sie wird vom Hauptfenster aufgerufen.
void Board::RescaleFast()
{
  //Erzeugen der skalierten Pixmap. Grundlage ist die aktuelle Pixmap (this->Pixmap)
  setPixmap(this->pixmap().scaled(m_view->width(),m_view->height(),
                                  Qt::KeepAspectRatio,Qt::SmoothTransformation));

  PlaceInView(); //Spielbrett in der Fenstermitte zentrieren
  DrawGrid(); //Rahmen und Trennlinien neu skalieren und positionieren
  ImageReloadTimer.start(ReloadDelay); //Timer starten – löst bei Ablauf "RescaleClean" aus.
}



//Sauberes Skalieren. Die Methode wird bei Ablauf des Times aufgerufen.
void Board::RescaleClean()
{
  Erzeugen der skalierten Pixmap. Grundlage ist die in BoardImage gespeicherte Bilddatei
  this->setPixmap(QPixmap::fromImage(BoardImage).scaled(m_view->width(),m_view->height(),
                                          Qt::KeepAspectRatio,Qt::SmoothTransformation));

  PlaceInView(); //Spielbett in der Fenstermitte Zentrieren
  DrawGrid(); //Rahmen und Trennlinien neu skalieren und positionieren
  emit CleanRedrawDone(); //Event auslösen, das vom Tokenmanager erfasst wird.
}
```

In der Methode *DrawGrid*, welche von den Skalierungsmethoden aufgerufen wird, werden die Positionen und Größen der Trennstriche und des Rahmenrechtecks berechnet. Alle dafür nötigen Parameter wurden dem Objekt zuvor durch das Setzen eines Spielbrettbildes und das Definieren der Spielfeldeigenschaften mitgeteilt. Die Größe des Views kann die Methode über den Zeiger auf den View erhalten, welcher dem Spielbrettobjekt beim Erstellen übergeben wurde.

Das Rahmenrechteck hat noch eine weitere Funktion als nur das bloße Zeichnen eines Rahmens. Einmal skaliert und positioniert gibt es genau an, in welchem Bereich des Views sich die Spielfelder

befinden. Da auch die Anzahl der Reihen und Spalten bekannt ist, kann somit leicht die Position jedes einzelnen Feldes berechnet werden. Hierfür gibt es die Methode *getFieldRect*, deren Parameter zwei Integer-Variablen sind, die die Zeilen- und Spaltennummern bestimmen, und deren Rückgabewert ein Objekt des Typs *QRect* ist, welches genau die Position und Größe des über die Parameter ausgewählten Spielfeldes hat. Diese Methode wird später dazu benutzt, um die Spielsteine richtig zu skalieren und zu positionieren. Die Spielfeldkoordinaten werden mit 0 beginnend gezählt, wobei mit der Methode *setVerticalFieldNumberInversion* eingestellt werden kann, ob sich das Feld mit der Koordinate 0/0 links oben oder links unten befindet.

Für den umgekehrten Weg gibt es die Methode *DetermineFieldAtPos*. Sie bestimmt für eine ihr übergebene Position im View das entsprechende Spielfeld. Zusätzlich bestimmt sie, ob sich dieses Feld seit dem letzten Aufruf der Methode geändert hat. Die Anwendung der Methode ist das Auswerten der Mausinteraktionen. Da das Spielbrettobjekt von dem Objekt *QGraphicsPixmapItem* erbt, hat es auch eine virtuelle Methode mit dem Namen *mousePressEvent*, die vom Qt-Framwork bei einem Mausklick auf die Grafik im View aufgerufen wird. Diese Methode kann überladen werden, um eine eigene Verarbeitung der Mausklicks zu implementieren. Sie hat einen Parameter des Typs *QGraphicsSceneMouseEvent*, in dem ihr zusätzliche Informationen wie die auslösende Maustaste sowie der Ort des Klicks übergeben werden.

Wenn nicht nur die Tastendrücke, sondern auch die bloße Bewegung der Maus ausgewertet werden sollen, gibt es jedoch eine zusätzliche Schwierigkeit. Der MouseMouveEvent, der bei einer Mausbewegung ausgelöst wird, wird nämlich nicht an das Spielbrettobjekt, sondern an den View gesendet, beziehungsweise an den Viewport, der ein Teil des Views ist. Jedoch bietet das Qt Framework auch hierfür eine Lösung – nämlich das Verwenden eines so genannten Eventfilters. Ein Eventfilter ist eine Funktion mit dem Namen *eventFilter* die einen Zeiger auf ein *QObject* und einen auf einen *QEvent* als Parameter übernimmt und selbst einen Bool-Wert zurückliefert. Hat ein *QObject* eine solche Funktion, kann es einem anderen *QObject* durch Aufrufen dessen *installEventFilter*-Methode als Eventfilter zugeordnet werden. Ist dies geschehen, rufen künftig alle Events, die ansonsten eine entsprechende Methode des Objekts mit dem Filter aufrufen würden, stattdessen die Filterfunktion auf. In dieser kann der Event dann zunächst verarbeitet werden. Je nachdem, ob die Filterfunktion *true* oder *false* zurückliefert, wird der Event anschließend an die Methode, an die er ursprünglich adressiert war, weitergeleitet oder nicht. Es sind auch Filterkaskaden möglich, also dass Objekte, die selbst Eventfilter für andere Objekte sind, ebenfalls einen Eventfilter haben.

Listing 5 zeigt Mauseventverarbeitung und das Installieren eines Eventfilters. Auch hier wurden die Methoden zur leichteren Verständlichkeit teilweise gekürzt.

Listing 5: Eventfilter und Mausevents

```
//Deklaration der Klasse Board (nur ausschnittweise)
class Board : public QObject, private QGraphicsPixmapItem
{
  Q_OBJECT //Wichtig, damit das Eventsystem genutzt werden kann
public:
 Board(QGraphicsView* viewPtr); //Konstruktor

signals: //Signale, die bei Mausinteraktionen ausgelöst werden
  void MouseOverFieldChanged(int Row, int Column);
  void MouseClickedOnBoard(int Row, int Column, Qt::MouseButton Button);

protected:
  virtual void mousePressEvent ( QGraphicsSceneMouseEvent * e );
  bool eventFilter(QObject* object, QEvent* event);
}



//Konstruktor der Klasse Board (vereinfacht)
Board::Board(QGraphicsView *viewPtr): m_view(viewPtr)
{
  m_view->viewport()->installEventFilter(this); //Dieses Objekt zum Eventfilter machen
}



//Event, das ausgelöst wird, wenn die Maus auf dem Spielbrett gedrückt wurde
void Board::mousePressEvent ( QGraphicsSceneMouseEvent * e )
{
  if(this->boundingRect().contains(e->pos())==false){ //Fand der Klick im Spielfeld statt?
      return;
  }
  DetermineFieldAtPos(e->pos()+this->pos()); //Bestimme das angeklickte Feld
  emit MouseClickedOnBoard(m_MouseOverRow, m_MouseOverColumn, e->button());
}



//Eventfilter vom View. Wird u. A. aufgerufen, wenn die Maus im View bewegt wird
bool Board::eventFilter(QObject* object, QEvent* event)
{
  if(event->type()==QEvent::MouseMove){//Ist eine Mausbewegung der Auslöser?

    //Bestimme das Feld unter der Maus – hat es sich geändert, löse einen Event aus
    if(DetermineFieldAtPos(static_cast<QMouseEvent*>(event)->posF())){
      emit MouseOverFieldChanged(m_MouseOverRow, m_MouseOverColumn);
    }
    return true; //Event nicht mehr weiterleiten
  }
  return false; //Keine Mausbewegung – Event an den View weiterleiten
}
```

Die Token

Der zweite Teil des Frameworks besteht aus einer Klasse für die Spielsteine sowie einer Klasse zu deren Verwaltung. Die Spielsteinklasse, die den Namen „Token“ trägt, erbt von der Klasse *QGraphicsPixmapItem* und ist damit, ebenso wie das Spielbrett, in der Lage, eine Grafik auf der Scene darzustellen. Die Klasse ist sehr klein gehalten und ihre cpp-Datei ist keine 60 Zeilen lang. Die beiden wichtigsten Methoden sind *PlaceScaledInRectFast* und *PlaceScaledInRectClean*, welche beide die Spielsteingrafik so skalieren und positionieren, dass sie ein der Methode übergebenes Rechteck genau ausfüllen. Wie schon im Zusammenhang mit dem Spielbrett beschrieben, benutzt die Fast-Methode dabei eine im Vergleich zur Clean-Methode schnellere, aber unsauberere Skalierung und ist nur für das Ermöglichen einer flüssigen Änderung der Fenstergröße gedacht. Weitere Methoden der Token-Klasse ermöglichen es, den Spielstein an einer Position zu zentrieren, seinen Typ zu ändern

sowie sein Bild, welches stets mit dem Typ verknüpft ist, zu laden. Die Klasse hat einige Membervariablen, auf die man von außen zugreifen und die man für die Verwaltung des Spielsteines verwenden kann. Es gibt zwei Integer-Variablen, die die aktuelle Position auf dem Spielfeld speichern, sowie zwei, die für ein mögliches Ziel einer Bewegung gedacht sind. Des Weiteren gibt es eine Fließkommazahl, die bei der Bewegung des Spielsteins verwendet wird. Letztendlich enthält sie noch einen Zeiger auf eine Struktur mit der Bezeichnung *TokenType*, in der Werte hinterlegt sind, die für alle Spielsteine einer bestimmten Sorte gleich sind. Dies ist zum einen ein Zeiger auf ein *QImage*, in dem die Grafik des Spielsteins hinterlegt ist, und zum anderen eine Variable, die den Layer des Spielsteines festlegt. Dieser bestimmt bei sich überlappenden Grafiken, welche vor und welche hinter der jeweils anderen gezeichnet wird. Zusätzlich enthält jede Struktur eine eindeutige ID-Nummer, eine Nummer, die den Spieler, der die Figur kontrolliert bestimmt, sowie eine Fließkommavariable mit dem Namen *ScaleFactor*. Diese wird von den *PlaceScaledInRect*-Methoden ausgewertet und kann, wenn sie ungleich 1 ist, dafür sorgen, dass die Grafik etwas kleiner oder größer als das Spielfeld skaliert wird. Gerade Letzteres ist nützlich, weil es dadurch zum Beispiel möglich ist, eine Grafik zu schaffen, die das ihr zugeordnete Spielfeld umrahmt. Die Klasse eignet sich nämlich nicht nur zum Darstellen von Spielsteinen, sondern kann für alle Grafikobjekte verwendet werden, die auf den Spielfeldern platziert werden sollen. Dies können auch Markierungen für die ausgewählten Spielsteine oder die aktuellen Zugmöglichkeiten sein. Es wurde abgewogen, ob es sich dabei nicht um eine unsaubere Bündelung verschiedener Aufgaben bei einem Objekt handelt. Jedoch ist es der Übersichtlichkeit nicht zuträglich, wenn eine Klasse, deren Quellcode auf eine Bildschirmseite passt, zertrennt wird, zumal der Übergang zwischen Spielstein und Markierung tatsächlich fließend sein kann.

Abbildung 6 zeigt verschiedene Spielsteine und Marker. Da man auch Bilddateien mit Alphakanal, wie z. B. PNG-Dateien, als Marker verwenden kann, gibt es sehr flexible Gestaltungsmöglichkeiten. Das blaue Feld oben rechts zeigt z. B. einen Marker der aus einem halbtransparenten blauen Rechteck besteht und dessen Layerwert höher ist als der des dahinter angezeigten Spielsteins. Der rote Rahmen unten rechts zeigt wiederum einen Marker, bei dem die *ScaleFactor*-Variable einen Wert größer als 1 hat. Daher ist die angezeigte Grafik etwas größer als das darunterliegende Spielfeld.

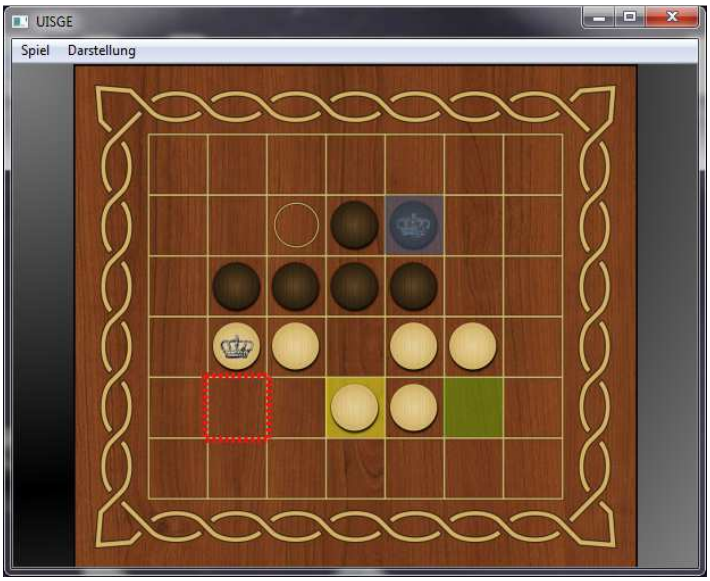

Abbildung 6: Spielsteine und Marker

Der Tokenmanager

Die Klasse Tokenmanager dient zum Verwalten der Token-Objekte. Ihr Hauptelement ist eine Liste, die Zeiger zu allen Tokenobjekten enthält, zudem enthält sie etliche Methoden, um auf die Einträge in dieser Liste zuzugreifen. Für die Liste wurde kein List-Container aus der Standard Template Libary (STL), sondern eine *QList* verwendet. Die Qt-Eigenen Templateklassen sind jedoch kompatibel zur STL und lassen sich genauso bedienen. Der Tokenmanager hat noch eine weitere Liste, die Zeiger zu allen *TokenType*-Strukturen enthält. Das Erstellen neuer *TokenType*-Strukturen oder Token-Objekte wird ausschließlich mit Hilfe des Tokenmanagers durchgeführt. Für erstere gibt es die Methode *AddTokenType*, der man einen String mit dem Dateinamen der Grafik übergibt, sowie drei Integer, die die Spielernummer, den *Layer* und den *ScaleFactor* der Spielsteine dieses Typs bestimmen. Die drei Integer-Parameter sind dabei optional, da hier Defaultparameter verwendet wurden. Die Methode liefert einen Integer zurück, der eine eindeutige Kennung der erstellten *TokenType*-Struktur darstellt und auch in die Struktur eingetragen wird. Dieser Wert sollte von der aufrufenden Funktion gespeichert werden, da er zum Erstellen eines Tokens mit dem entsprechenden Typ nötig ist. Zwar werden die Kennungen aufsteigend vergeben, so dass z. B. der *TokenType*, der als 3. erstellt wurde, immer die Kennnummer 2 hat, jedoch ist es aus Gründen der Übersichtlichkeit zu empfehlen, die Kennungen der einzelnen Spielsteinsorten in einer sinnvoll benannten Variable zu speichern. Nachdem man einen *TokenType* definiert hat, kann man den Tokenmanager dafür verwenden, um einzelne Tokens dieses Typs zu erstellen. Dafür bietet dieser die Methode *AddToken*, welcher die Kennung des *TokenType* übergeben werden muss. Optional kann man der Methode auch noch eine Zeilen- und Spaltennummer übergeben, was dazu führt, dass der neu erstellt Token direkt auf das entsprechende Spielfeld gesetzt wird. Gibt man nichts an, landet der Token auf dem Feld mit den Koordinaten 0/0. Die Methode *AddToken* erstellt den Token, trägt ihn in die Liste des Tokenmangers ein und liefert einen Zeiger auf den Token zurück. Dieser kann zum Beispiel dazu verwendet werden, um den Token sichtbar oder unsichtbar zu machen oder auf seine öffentlichen Methoden zuzugreifen, um zum Beispiel seine momentane Position auszuwerten.

Die Tokenmanger-Klasse hat einige Methoden, die zur Verwaltung der Token benutzt werden können. So kann man z. B. die Gesamtanzahl aller Token oder auch die Anzahl der Token eines gewissen Typs, Spielers oder auf einem bestimmten Feld abfragen. Man hat auch verschiedene Möglichkeiten, einen Zeiger auf die verwalteten Token zu bekommen. Beispielsweise gibt es die Methode *getTokenOnField*, die einen Zeiger auf den Token auf einem bestimmten Feld zurückgibt. Optional kann man der Methode noch eine Nummer übergeben, falls sich mehr als ein Token auf dem angegebenen Feld befindet. Zusammen mit der Methode *TokensCountOnField*, die die Anzahl der Token auf einem Feld ermittelt, kann man in einer Schleife alle Token auf einem Feld erhalten und mit ihnen eine Aktion durchführen oder prüfen, ob einer davon eine bestimmte Eigenschaft erfüllt. Diesen Mechanismus gibt es nicht nur für ein bestimmtes Spielfeld, sondern auch für einen *TokenType* oder eine Spielernummer. Im Gegensatz zu den TokenTypes kann man Token-Objekte auch löschen. Dies verändert jedoch die Position der anderen Token in der Liste. Daher gibt es auch eine Methode, mit der man die Listenposition eines Tokens anhand eines Zeigers auf diesen Token ermitteln kann. Zusammen mit der reziproken Methode, die anhand einer ihr übergebenen Nummer den entsprechenden Zeiger liefert, hat der Anwender somit eine sehr große Flexibilität beim Umgang mit den Token. Listing 6 zeigt beispielhaft die Initialisierung und Verwendung des Tokenmanagers.

Listing 6: Verwendung des Tokenmanagers

```
TheBoard = new Board(ui->MainView);
TheTokenManager = new TokenManager(TheBoard);

TheBoard->setImage("Spielbrettbild.jpg"); // Spielfeldgrafik laden

// Spielfeldbereich festlegen. 6 Zeilen und 5 Spalten
TheBoard->DefinePlayingField(QRect(100,100,500,300),6, 5);

// Spielsteintyp Bauer: Spieler = 0; Layer = 0; ScaleFactor = 1;
int TypBauer = TheTokenManager->AddTokenType("Bauer.jpg");

// Spielsteintyp König: Spieler = 1; Layer = 22; ScaleFactor = 1.5;
int TypKoenig = TheTokenManager->AddTokenType("Koenig.jpg",1,22,1.5);

TheTokenManager->AddToken(TypBauer);          // Bauer an Position [0/0] erstellen
TheTokenManager->AddToken(TypBauer,3,5);      // Bauer an Position [3/5] erstellen
TheTokenManager->AddToken(TypKoenig);         // König an Position [0/0] erstellen

TheTokenManager->TokensCountForType(TypBauer);  // Anzahl der Bauern bestimmen (2)
TheTokenManager->TokensCountOnField(3,5); // Anzahl an Tokens auf dem Feld [3/5] bestimmen (1)

//TempTokenPtr wird zu einem Zeiger auf den Bauer auf Feld [3/5].
Token* TempTokenPtr = TheTokenManager->getTokenOnField(3,5);

TheTokenManager->DeleteToken(TempTokenPtr); // Der Bauer auf Feld [3/5] wird entfernt.
```

Der Tokenmanager kümmert sich auch um die Neuplatzierung der Tokens nach und während eines Skaliervorgangs. Dafür hat er die beiden Methoden *ReplaceAllTokensFast* und *ReplaceAllTokensClean*, die jeweils nach der Neuskalierung des Spielbretts aufgerufen werden und in denen jeder Token mit der entsprechenden PlaceScaledInRect-Methode skaliert und auf seinem Spielfeld platziert wird.

Die Bewegung der Spielsteine wird ebenfalls mit Hilfe der Tokenmanager-Klasse realisiert. Dafür wird in zwei Schritten vorgegangen. Als erstes wird mit der Methode setTokenMoveDestination für alle Token, die bewegt werden sollen, ein Zielfeld festgelegt. Dabei prüft der die Methode, ob sich der Token bereits auf diesem Zielfeld befindet. Ist dies nicht der Fall, wird die Membervariable *m_MovePercent* des Tokens auf 0 gesetzt. Nachdem alle Zielkoordinaten gesetzt wurden, wird die Bewegung durch Aufruf der Methode *MoveTokens* gestartet. In dieser wird ein Timer gestartet, der fortan alle 20 Millisekunden die Methode *MoveTimestep* aufruft, in der wiederum für alle Tokens deren Membervariable *m_MovePercent* kleiner als 100 ist, die Methode *AdvanceToken* aufgerufen wird. In dieser wird die Start- und Zielposition des Tokens bestimmt und der Token dann an eine Position zwischen diesen beiden Punkten positioniert, die mittels linearer Interpolation bestimmt wird. Dies geschieht in Abhängigkeit von der Variablen *m_MovePercent*, wobei der Wert 0 für die Startposition und der Wert 100 für die Zielposition steht. Die Variable *m_MovePercent* wird zudem bei jedem Aufruf erhöht, was zu einer kontinuierlichen Bewegung führt. Der Abstand zwischen dem Start- und dem Zielpunkt hat dabei Einfluss auf die Erhöhung, was nötig ist, um eine stets gleiche Bewegungsgeschwindigkeit zu erreichen. Die Geschwindigkeit kann mit der Methode *setMovingSpeed* eingestellt werden. Während der Bewegung wird der Layer der Tokens erhöht, womit erreicht wird, dass sich die bewegten Token für den Betrachter über die stehenden hinweg und nicht unter ihnen hindurch bewegen. Sobald alle Token ihren Zielort erreicht haben, wird der Timer wieder angehalten und es wird ein Event ausgelöst, der von anderen Programmteilen ausgewertet werden kann. Über die Methode *MovingInProgress* kann zudem jederzeit abgefragt werden, ob sich Token in Bewegung befinden.

Entwicklung eines UISGE-Spiels mit dem Brettspiel-Framework

Um die Bedienbarkeit des Framewoks und seine Eignung zum Entwickeln zu testen und gleichzeitig eine Anwendung zu schaffen, die alle Funktionalitäten des Frameworks präsentiert, wurde zunächst ein komplett eigenständiges UISGE-Spiel von Grund auf programmiert. Der eigentliche Teil der Aufgabenstellung ist zwar, die bestehende Konsolenanwendung mit einer grafischen Benutzerschnittstelle auszustatten, jedoch sprechen einige Gründe dafür, auch den anderen Weg zu gehen und ein Spiel zu entwickeln, das die Möglichkeiten des Frameworks konsequent nutzt. Denn dieses kann zum Beispiel die Verwaltung der Spielsteine übernehmen und somit beim Entwickeln des Spiels viel Arbeit und Quellcode sparen. Die bestehende Konsolenanwendung hat im Gegensatz dazu eine eigene Spielsteinverwaltung, welche mit der des Frameworks synchronisiert werden muss. Um zu prüfen, wie schnell und einfach die Implementierung mit Hilfe des Frameworks machbar ist, wurde zunächst eine Neuentwicklung gemacht. Später konnten einige Teile des Codes der Neuentwicklung auch dazu verwendet werden, den Quellcode der Konsolenanwendung mit dem Grafikframework zu verbinden.

Die Neuentwicklung besteht aus zwei Modulen, deren Implementierung zusammen nicht einmal 500 Zeilen lang ist. Die Klasse *GameUisge* steuert den Spielablauf und nimmt Befehle des Nutzers entgegen und die Klasse *RuleChecker* prüft mögliche Züge auf ihre Vereinbarkeit mit den Spielregeln. Die Spielregeln sind auf der dieser Arbeit beigefügten CD zu finden und werden hier daher nur ausschnittweise beschrieben. Gemeinsam haben die Klassen etwas mehr Funktionalität als die Konsolenanwendung. So werden die möglichen Züge eines ausgewählten Spielsteins im Voraus ermittelt und können dem Spieler als Hilfe angezeigt werden. Auch der letzte Zug des Gegenspielers kann mit Markierungen auf dem Spielfeld sichtbar gemacht werden. Wirklich spielrelevante Neuerungen sind eine Undo-Funktion, mit der man Spielzüge zurücknehmen kann, und die Beseitigung einer kleinen Regelungenauigkeit. Die ursprünglichen Uisge-Regeln machen nämlich keine Aussage dazu, wie verfahren werden soll, wenn ein Spieler in einer Situation ist, in der er keinen einzigen Stein bewegen kann. Diese Situation ist selten, jedoch nicht unmöglich. Die Konsolenapplikation geht darauf nicht explizit ein und hängt sich somit praktisch auf. Sie verlangt, dass der Spieler, der grade an der Reihe ist, zieht, lehnt aber jeden Zugversuch aufgrund des Regelwerks ab. In der Neuentwicklung wird hingegen der Code, der auch zum Anzeigen der gültigen Spielzüge verwendet wird, dazu genutzt, um zu überprüfen, ob der Spieler, der am Zug ist, überhaupt einen Stein bewegen kann. Ist dies nicht der Fall, verliert er das Spiel.

Bei Programmstart wird eine Instanz der Klasse *GameUisge* erstellt, die in ihrem Konstruktor zunächst einige Initialisierungen vornimmt. Sie verbindet eigene Eventempfängerfunktionen (Slots) mit eventauslösenden Funktionen (Signale) der Frameworks, zum Beispiel die beiden Signale, die von der Board-Klasse ausgelöst werden, wenn die Maus bewegt oder geklickt wird. Anschließend wird das Spielbrett durch Setzten einer Grafik und Bestimmen der Spielfläche und der Zeilen- und Spaltenzahl konfiguriert. Im nächsten Schritt, werden alle benötigten TokenTypes definiert. Dabei werden ihre Grafik, der kontrollierende Spieler und ihr Layer definiert. Es wird für alle vier Spielsteintypen ein *TokenType* benötigt sowie für verschiedene Markierungen. Dies ist eine Markierung für das Feld unter der Maus, eine für den ausgewählten Spielstein, zwei um Start- und Zielfeld des letzten Zuges anzuzeigen sowie eine zur Anzeige der möglichen Züge. Anfangs wurden sogar zwei verschiedene Markierungen zur Anzeige von Bewegungs- und Sprungzügen verwendet. Da dies jedoch die Übersichtlichkeit eher verschlechterte, wurde später davon abgesehen. Nachdem die

TokenTypes definiert sind, werden von den Token, deren Anzahl unveränderlich ist, die entsprechenden Instanzen angelegt und Zeiger zu diesen gespeichert. Dies sind die Markierungen für den letzten Zug, den ausgewählten Spielstein und das Feld unter der Maus. Anschließend wird eine neue Instanz der *RuleChecker*-Klasse angelegt. Ihr werden die Identifikationsnummern der TokenTypes der Spielsteine übergeben, welche diese in einer internen Liste speichert. Dabei wird der *RuleChecker*-Instanz auch mitgeteilt, ob die jeweilige Nummer zu einem leeren Stein oder einem Kronestein gehört. Der *RuleCheker*, der über einen Zeiger auf den *TokenManager* zugreifen kann, ist damit in der Lage, selbständig das Spielfeld zu analysieren und festzustellen, wo sich welche Steine befinden und was für Züge diese ausführen dürfen.

Nach der Initialisierung wird die Methode *SetUpNewGame* ausgeführt, welche ein neues Spiel startet. Diese Methode kann auch über das Eventsystem aufgerufen werden. Bei Programmstart wird sie mit dem Event verbunden, das beim Klicken auf den Menüpunkt „Neu“ in dem Untermenü „Spiel“ der Menüleiste ausgelöst wird. Somit wird bei jedem Anklicken dieses Menüpunktes ein neues Spiel gestartet. Das Erstellen eines verzweigten Menüs in der oberen Menüleiste lässt sich einfach mit dem Formulareditor bewerkstelligen. Dort kann auch der Name des Events festgelegt werden, mit dem man ihn dann mit der *connect*-Funktion mit einem Slot verbinden kann. In der Methode *SetUpNewGame* werden zunächst alle Spielsteine gelöscht, wobei bei ihrem ersten Aufruf natürlich noch keine Spielsteine existieren. Zum Löschen wird die kleine Hilfsfunktion *DeleteAllTokensOfType* verwendet, die die Möglichkeiten des TokenManagers nutzt, um alle Token eines Typs zu entfernen. Diese wird auch dazu verwendet, um die Markierungen für die möglichen Züge wieder zu entfernen, wenn sich der ausgewählte Spielstein ändert. Nach dem Löschen werden neue Spielsteine an den Positionen der Grundaufstellung erstellt, der Zugzähler wird auf 0 gesetzt und die Markierungen des letzten Zuges werden versteckt. Zusätzlich wird eine Liste geleert, in der alle getätigten Züge gespeichert werden und die zur Realisierung der Undo-Funktion nötig ist. Die Methode ruft keine weiteren Funktionen auf. Sobald sie ihren Endpunkt erreicht hat, ist der Programmablauf nicht mehr linear, sondern rein eventgetrieben. Dabei gibt es, abgesehen von den Aktionen, die der Benutzer über die Menüleiste ausführen kann, lediglich zwei Events. In der Methode *MouseOverFieldChanged*, die durch das Spielbrett ausgelöst wird, wird lediglich die Position der Markierung des Feldes unter der Maus geändert und die Markierung gegebenenfalls sichtbar oder unsichtbar gemacht. In der Methode *MouseClickedOnBoard*, die ebenfalls durch das Spielbrett ausgelöst wird, werden Mausklicks verarbeitet und somit die eigentliche Spielsteuerung durchgeführt. Der Methode werden die Koordinaten des angeklickten Feldes und der auslösende Mausbutton übergeben. Handelt es sich um die linke Maustaste wird sie aktiv, ansonsten beendet sie sich sofort wieder. Da das Spiel also ausschließlich mit der linken Maustaste gesteuert wird, kann es auch problemlos mit einem Touchscreen bedient werden, da bei diesem eine Berührung üblicherweise einen Klick mit der linken Maustaste erzeugt. Wurde der Event mit der linken Maustaste ausgelöst, prüft die Methode zunächst, ob sich an der übergebenen Position ein Spielstein befindet. Ist dies der Fall, und gehört der Spielstein dem Spieler der gerade am Zug ist, wird dieser ausgewählt. Das heißt, der Markierer, der den ausgewählten Spielstein hervorhebt, wird auf dieses Feld versetzt und sichtbar gemacht. Ein Ausnahmefall ist, wenn der Stein bereits ausgewählt war. Ist dies der Fall, wird die Auswahl zurückgenommen und der Auswahlmarkierer wird unsichtbar gemacht. Jedes Mal, wenn ein Spielstein markiert wird, werden die Möglichen Züge des Spielsteins ermittelt und durch Erstellen von Markern auf den entsprechenden Feldern angezeigt. Sobald sich der ausgewählte Stein ändert, werden alle diese Markierungen wieder entfernt. Zum Prüfen der möglichen Züge wird mit Hilfe des RuleCheckers für alle 12 theoretisch möglichen Züge (8 angrenzende Felder + 4 Sprungmöglichkeiten)

die Machbarkeit ermittelt. Der *RuleChecker* berücksichtigt dabei selbstständig die Art des Spielsteins sowie die Position aller anderen Spielsteine. Wie er dabei genau vorgeht, wird später erklärt.

Fand der Mausklick auf einem freien Feld statt und war gerade ein Spielstein markiert, wird die Methode *TryMoveToField* aufgerufen, die, wie ihr Name sagt, versucht, einen Zug auf das angeklickte Feld auszuführen. Das Startfeld der Bewegung ist dabei die aktuelle Position des Auswahlmarkers. Zur Prüfung der Regelkonformität wird natürlich wieder das *RuleChecker*-Objekt verwendet. Ist der Zug gültig, werden die Zielkoordinaten des Spielsteins an der Position des Auswahlmarkers auf das angeklickte Feld gesetzt und die Bewegungsanimation wird gestartet. Anschließend werden die Markierungen, die den letzten Zug anzeigen, auf die Start- und Zielposition gesetzt und der Zug wird, um ein späteres Zurücknehmen zu ermöglichen, gespeichert. Dafür gibt es die Struktur *MoveData*, die vier Integer-Variablen enthält, mit der Start- und Zielposition eines Zuges gespeichert werden können. Es wird also eine neue Struktur dieses Typs angelegt, mit entsprechenden Werten gefüllt und einer Liste, die Teil des *GameUisge*-Objekts ist, hinzugefügt. Bei begrenztem Speicherplatz müsste man noch theoretisch sicherstellen, dass die Liste nicht zu groß wird. Da die einzelnen Objekte aber recht klein sind, lassen sich auch auf einem schwachen System theoretisch tausende Züge speichern. Zudem wird die Liste bei Beginn eines neuen Spiels geleert.

Nachdem der Zug gestartet ist, beendet sich die Methode und es passiert zunächst erst einmal nichts. Insbesondere werden während der Bewegung auch keinerlei Mauseingaben akzeptiert. Sobald die Bewegung abgeschlossen ist, löst der *TokenManager* ein Event aus, das zum Aufruf der Methode *MovingDone* führt. In dieser wird geprüft, ob der Spieler das Spiel gewonnen hat. Dafür wird zum einen geprüft, ob er über 6 Kronesteine verfügt, und zum anderen, ob der Gegenspieler zu mindestens einem Zug fähig ist. Hat der Spieler gewonnen, wird eine entsprechende Meldung ausgegeben, ansonsten kommt der nächste Spieler an die Reihe und das Spiel wird fortgesetzt.

Da man im Uisge keine Spielsteine schlagen kann, ist die bereits erwähnte Undo-Funktion ganz einfach. Sie führt einfach den letzten Zug mit vertauschten Start- und Zielkoordinaten durch. Dabei wird in diesem Fall auf eine Bewegungsanimation verzichtet und der Stein wird direkt platziert. Zusätzlich werden die Markierungen für den letzten Zug auf die Koordinaten des Zuges vor dem letzten Zug platziert oder, sofern es keinen vorletzten Zug gab, unsichtbar gemacht. Nach Zurücknehmen eines Zuges wird der entsprechende Eintrag aus der Liste gelöscht. Auch die Undo-Funktion wird über ein Signal, das von einem Button der Menüleiste erzeugt wird, aufgerufen.

Der algorithmisch interessanteste Teil des Programms findet in der Klasse *RuleChecker* statt, die die Konformität eines Zuges mit dem Regelwerk prüft. Dabei geht sie schrittweise vor, um alle Regeln in Betracht zu ziehen. Zuerst prüft sie, ob sich auf dem ihr übergebenen Startfeld überhaupt ein Spielstein befindet und ob dieser Stein theoretisch die Bewegung zum Zielfeld durchführen könnte. Dabei wird auch darauf eingegangen, ob es sich um einen leeren Stein oder einen Kronestein handelt. Ist der Zug ein Sprungzug, wird zusätzlich ermittelt, ob sich auf dem übersprungenen Feld ein Stein befindet. Dabei wird ausgenutzt, dass die Koordinaten des übersprungenen Feldes stets der arithmetische Mittelwert der Koordinaten des Start- und Zielfeldes sind. Sind diese Voraussetzungen erfüllt, beginnt die zweite Phase des Tests. Hierfür bedient sich der *RuleChecker* einer Struktur mit dem Namen *FieldWithToken*, die die Koordinaten eines Feldes sowie einen Bool-Wert Namens *Visited* speichern kann, dessen Aufgabe später erklärt wird. Der *RuleChecker* nutzt nun den *TokenManager*, um die Koordinaten aller Spielsteine abzufragen und jeweils in einer Struktur zu speichern, die auch in einer Liste verwaltet werden kann. Da der *RuleChecker*, wie bereits

beschrieben, eine Liste mit den Identifikationsnummern aller *TokenTypes* der Spielsteine hat, kann er dies in einer simplen Schleife erledigen. Hat er dies erledigt, hält er eine Liste aller Felder mit Spielsteinen, in der er den geplanten Zug simuliert. Dafür sucht er den Eintrag mit den Koordinaten des ihm übergebenen Startfeldes und setzt diese auf die Koordinaten des Zielfeldes. Diese virtuelle Spielsituation wird jetzt mehreren Tests unterzogen, um zu prüfen, ob sie erlaubt ist.

Zuerst wird überprüft, ob es Felder gibt, auf denen mehr als ein Spielstein ist. Ist dies der Fall, war das Zielfeld nicht frei und der Zug ist nicht erlaubt. Dies könnte man jedoch auch leicht ohne den komplizierten Umweg mit der Liste prüfen. Anschließend wird geprüft, ob sich ein Stein außerhalb des Spielfeldes befindet, was den Zug ebenfalls unmöglich macht. Auch dafür ist die Liste nicht zwingend notwendig. Der letzte Test jedoch, der das Zusammenhängen aller Spielsteine überprüft, lässt sich mit Hilfe der Liste sehr elegant lösen. Dafür wird der Floodfill- oder Flutfüllungsalgorithmus verwendet, der z. B. in der Computergrafik dazu verwendet wird, Flächen zusammenhängender Pixel einzufärben. Es handelt sich dabei um einen rekursiven Algorithmus, der an einer beliebigen Stelle innerhalb der Fläche gestartet werden kann und der zuverlässig alle Elemente dieser Fläche erreicht. Die Implementierung ist in Listing 7 gezeigt.

Zunächst werden alle Felder als nicht besucht markiert, wobei die Zahl der besuchten Felder in einer Membervariablen des *RuleChecker*-Objekts gespeichert ist, also von allen seinen Methoden verändert werden kann. Anschließend wird der Floodfillalgorithmus an der Position des ersten Felds der Liste gestartet. Dieser prüft nun für die ihm übergebene Position, ob sich an ihm ein Spielstein befindet und dieser noch nicht als besucht markiert ist. Dafür geht er einfach die Liste aller Felder durch. Wurde der Algorithmus fündig, markiert er das entsprechende Feld als besucht und ruft sich selbst mit den benachbarten Koordinaten des gefunden Steins auf. Ist auf einer dieser Nachbarkoordinaten ebenfalls ein noch nicht als besucht markierter Stein, setzt sich der Algorithmus fort. Dies geschieht so lange, bis der Algorithmus keine neuen Felder mehr findet. Jedes Mal, wenn der Algorithmus ein neues besetztes Feld als besucht markiert, erhöht er auch den Zähler für die Anzahl der besuchten Felder. Nachdem der Algorithmus durchgelaufen ist, kann anhand dieses Zählers einfach überprüft werden, ob er alle 12 Steine erreicht hat, also alle Steine entweder horizontal oder vertikal miteinander verbunden sind. Ist dies der Fall, ist der Spielzug regelkonform. Wie alle rekursiven Algorithmen birgt der Floodfillalgorithmus potentiell die Gefahr, einen Stack overflow, also einen Überlauf des Stapelspeichers zu verursachen. Daher ist es gefährlich, ihn einzusetzen, wenn man nicht abschätzen kann, wie häufig er sich selbst aufrufen wird. Will man zum Beispiel in einem Bild, das theoretisch Millionen von Pixeln haben kann, eine Fläche mit dieser Methode einfärben, sollte man Vorkehrungen treffen und wenn möglich lieber einen iterativen als einen rekursiven Algorithmus wählen. Man kann auch die Anzahl der Rekursionen zählen und den Algorithmus abbrechen, wenn diese zu groß wird. Da es jedoch nur 12 Spielsteine gibt, kann der Algorithmus problemlos eingesetzt werden, da spätestens der 13. Unteraufruf keine neuen Steine finden kann.

Abschließend wurden noch einige kosmetische Funktionen implementiert. So kann die Anzeige des letzten Zuges und der möglichen Züge oder die Hervorhebung des Feldes unter der Maus über den Menüpunkt Darstellung in der Menüleiste eingestellt werden. Auch wurde eine Funktion geschrieben, die ein Fenster in der Mitte des Hauptfensters anzeigt, in dem ein Text, den man der Funktion übergeben kann, angezeigt wird. Dies wird zur Anzeige der Spielregeln, der Siegesmeldung und der Programminformation verwendet. Für all dies wurden die Dialogfunktionen des Qt-Frameworks verwendet, die man mit Hilfe des Formulareditors schnell und einfach konfigurieren

kann. Als besonders praktisch stellte sich dabei das Dialogobjekt *QTextBrowser* heraus. Dieses ist ein Textfeld, das Text mit HTML-Syntax interpretieren und darstellen kann. Der Funktion zur Darstellung der Nachrichtenfenster kann also ein String übergeben werden, der alle Formatierungsvorgaben bereits enthält. Dies führt bei kleinen Projekten zu sehr übersichtlichem Code, weil der Text der Meldung direkt in den Quellcode geschrieben und auch dort formatiert werden kann.

Alles in allem kann festgestellt werden, dass die Entwicklung eines Brettspiels mit dem entwickelten Framework sehr leicht von der Hand geht und zu zufriedenstellenden Ergebnissen führt. Die Implementierung der wesentlichen Komponenten hat lediglich einen Tag benötigt, während ein zweiter Tag dafür verwendet wurde, den Code zu überarbeiten und zusätzliche Funktionen, wie die beschriebenen Grafikoptionen, einzubauen.

Listing 7: Prüfung der Spielsteinanordnung mit Floodfillalgorithmus

```
//Struktur, die besetztes Feld repräsentiert (in Datei rulechecker.h)
struct FieldWithToken
{
  int Row;
  int Column;
  bool Visited;
};



//Implementierung des Floodfillalgorithmus in der Datei rulechecker.cpp
void RuleChecker::RecursiveFloodFill4Check(int Row, int Column)
{
  //Prüfen, ob Feld in der Liste enthalten und noch nicht besucht
  for(int i = 0; i<ListOfFields.size(); i++){
    if((ListOfFields[i].Row == Row)&&(ListOfFields[i].Column == Column)&&
      (ListOfFields[i].Visited == false)){

        ListOfFields[i].Visited = true; //Feld als besucht markieren
        FieldsVisited++;                //Zähler für besuchte Felder erhöhen

        //Algorithmus für alle horizontal und vertikal angrenzenden Felder ausführen
        RecursiveFloodFill4Check(Row+1, Column);
        RecursiveFloodFill4Check(Row-1, Column);
        RecursiveFloodFill4Check(Row, Column+1);
        RecursiveFloodFill4Check(Row, Column-1);
    }
  }
  return;
}


//Initialisierung und Auswertung des Floodfill-Algorithmus
bool RuleChecker::CheckConnection()
{
  //alle Felder als noch nicht besucht markieren
  FieldsVisited = 0;
  for(int i = 0; i<ListOfFields.size(); i++){
    ListOfFields[i].Visited = false;
  }

  //Floodfillalgorithmus starten
  RecursiveFloodFill4Check(ListOfFields[0].Row,ListOfFields[0].Column);
  return (FieldsVisited == ListOfFields.size()); //Wurden alle Felder besucht?
}
```

Erstellung der Grafiken

Für die Erstellung der Grafiken wurde hauptsächlich das freie Vektorzeichenprogramm Inkscape verwendet. Das keltische Knotenmuster wurde mit dem kostenlosen Programm „Knotter“ erstellt, mit dem man komfortabel keltische Knotenmuster zeichnen und konfigurieren kann. Für die Holztextur des Spielfelds wurde eine Grafik aus dem Internet verwendet, die für nichtgewerbliche Nutzung freigegeben ist. Letztendlich wurde für das Spielbrett ein Rechteck mit dieser Holztextur erstellt und auf dieses ein Kästchenraster und das Knotenmuster gelegt. Die verwendeten Farben wurden dabei von denen der bereits vorhandenen Spielsteingrafiken kopiert. Für die Markierungen wurden entweder einfache farbige Rechtecke oder für die Anzeige des letzten Zuges ein Kreuz und ein Kreis verwendet. Die Farbigen Rechtecke wurden dabei halbtransparent gemacht, so dass die Holzstruktur des Spielbretts unter den Markierung zu sehen ist. Als Dateiformat wurde stets „png“ gewählt, da dieses Format Transparenz mittels Alphakanal unterstützt.

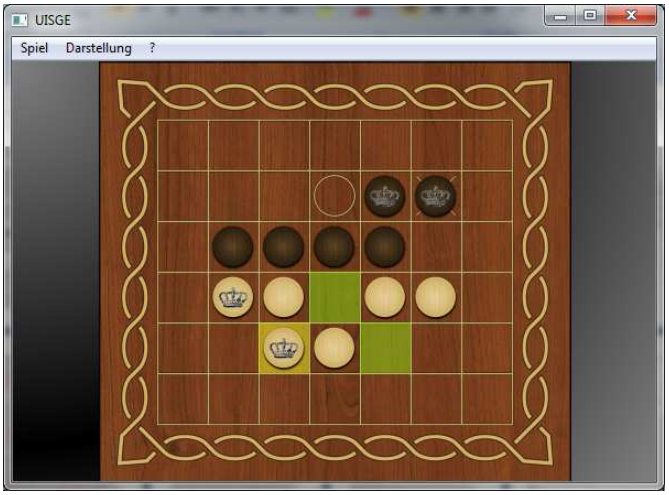

Abbildung 7: Screenshot des fertigen Spiels

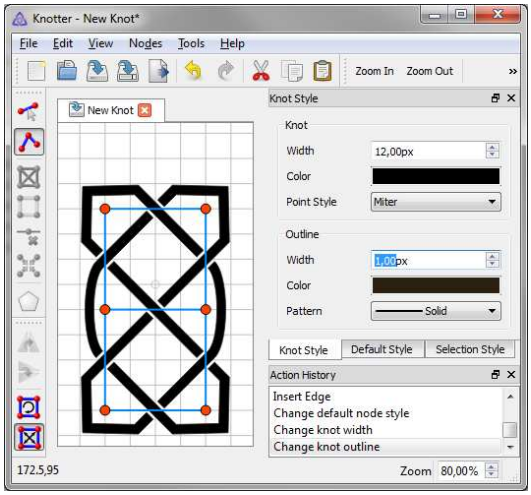

Abbildung 8: Screenshot des Programms "Knotter"

Portierung zu Linux

Die Entwicklung wurde in einem Windows-System getätigt. Da jedoch ausschließlich Funktionen des Qt-Frameworks genutzt wurden, ist das Programm auch ohne Änderungen unter Linux lauffähig. Dafür muss lediglich das Projekt in einem Linux-System mit dem Qt Creator geöffnet werden. Vorher sollte die user-Datei, welche die Daten des Entwicklungssystems enthält, gelöscht werden. Der Qt Creator erzeugt dann beim Öffnen des Projekts automatisch eine neue user-Datei mit den entsprechenden Einstellungen für das neue System. Man muss lediglich beachten, dass das Programm die Grafikdateien in seinem Ausführungsverzeichnis erwartet. Dieses Verzeichnis kann sich abhängig von den Projekteinstellungen im Qt Creator ändern. Als Hilfe kann man sich das aktuelle Verzeichnis auf der Debugkonsole ausgeben lassen.

Verzeichnisse

Quellen

[1] Martin, R. C. (2009). Clean Code. Heidelberg, München, Landsberg, Frechen, Hamburg: mitp-Verlag.

[2] Wikipediaartikel über den Floodfill-Algorithmus
http://de.wikipedia.org/wiki/Floodfill, o.V., o.J., Einsichtnahme: 5.12.2012

[3] Quelle der verwendeten Holztextur
http://www.photosof.org/view/wood_texture_background-other.html

Listings

Abbildungen