Softwarepraktikum über die Darstellung von Quaternionenfraktalen

im SS 2002 bei
Dr. Susanne Krömker und Dr. Michael Winckler

Tobias Dyckerhoff
tdyckerh@ix.urz.uni-heidelberg.de

David Reichert
david.reichert@web.de

Heidelberg, den 23. Juli 2002

1. Einleitung

Seit längerer Zeit zeigten wir schon Interesse an Quaternionenfraktalen und deren Darstellung. Bisher existieren hauptsächlich Programme, die die Darstellung mit Raytracing Algorithmen bewältigen. Dies hat allerdings den Nachteil, dass nachträgliches interaktives Erkunden des Fraktals unmöglich ist. Da eine Vorstellung von dreidimensionalen Objekten wohl am ehesten durch Anfassen bzw. Bewegen und Drehen vermittelt werden kann, machten wir es zur Aufgabe dieses Praktikums, ein interaktives Betrachtungsprogramm zur Darstellung von Quaternionenfraktalen zu schreiben. Als erste Anwendung sollte das Erstellen eines Pfades von Wegpunkten dienen, welcher dann mit Raytracing Methoden zu einer Kamerafahrt über die fraktale Oberfläche ausgebaut wird. Dieser Bericht soll einen Überblick über unsere Arbeit geben.

2. Theorie

2.1. **Die Hamilton'schen Quaternionen.** Um analoge Resultate zur Darstellung von Juliamengen im Körper der komplexen Zahlen $\mathbb{C}$ zu erhalten, stellt sich die Frage, welche algebraischen Strukturen, die (wie $\mathbb{C}$) den Körper der reellen Zahlen $\mathbb{R}$ enthalten, zu diesem Zwecke geeignet sind.
Man kann $\mathbb{C}$ als zweidimensionalen $\mathbb{R}$-Vektorraum konstruieren, wobei die multiplikative Struktur auf der Basis $\{1, i\}$ wie folgt zu definieren ist.

$$1i = i1 = i$$
$$i^2 = -1$$

Die so erhaltene Algebra erfüllt bekanntlich alle Körperaxiome. Nach dem Fundamentalsatz der Algebra besitzt jedes Polynom in $\mathbb{C}$ eine Nullstelle. $\mathbb{C}$ ist die (bis auf Körperisomorphie eindeutige) algebraisch abgeschlossene Hülle von $\mathbb{R}$ und es kann daher keine grösseren algebraischen Erweiterungskörper geben. Wenn man also versucht auf höherdimensionalen $\mathbb{R}$-Vektorräumen eine multiplikative Struktur zu definieren, so ist man gezwungen auf einige Eigenschaften eines Körpers zu verzichten.

Sei V ein $\mathbb{R}$-Vektorraum der Dimension 4 mit Basis $\{1, i, j, k\}$. Dann erhalten wir eine $\mathbb{R}$-Algebra, indem wir festlegen

$$i^2 = j^2 = k^2 = ijk = -1$$

Wir nennen diese Algebra die Hamilton'schen Quaternionen $\mathbb{H}$. Einfaches Nachrechnen bestätigt, daß $\mathbb{H}$ ein Schiefkörper, also ein Körper bis auf Kommutativität ist. Insbesondere ist also jedes $q \in \mathbb{H}$ invertierbar und für

$$q = q_0 + q_1 i + q_2 j + q_3 k \quad \text{mit} \quad q_i \in \mathbb{R}$$

setzen wir *das Konjugierte von q*

$$q^* := q_0 - q_1 i - q_2 j - q_3 k$$

sowie die *die Norm von q*

$$|q| := \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

Damit folgt

$$q^{-1} = \frac{q^*}{|q|} \quad \text{für} \quad q \neq 0$$

Der einzige algebraische “Verlust” gegenüber $\mathbb{C}$ ist also die Kommutativität , und mehr können wir nach obigen Betrachtungen auch nicht erwarten. Dies sollte genügen, um in $\mathbb{H}$ rechnen und analog wie in $\mathbb{C}$ Juliamengen in $\mathbb{H}$ definieren zu können.

2.2. **Juliamengen im Quaternionenraum.** Wir betrachten ein Polynom $f(X)$ mit Koeffizienten in den komplexen Zahlen $\mathbb{C}$. Für jedes $z \in \mathbb{C}$ entsteht eine Folge von komplexen Zahlen durch

$$z_0 = z \quad ; \quad z_{n+1} = f(z_n)$$

Wir bezeichnen den topologischen Abschluss der Menge

$$M = \{z \in \mathbb{C} \,|\, |z_n| \to \infty \quad (n \to \infty)\}$$

als Juliamenge von $f(X)$.

Wie im komplexen Fall, kann man so für jeden Punkt aus $\mathbb{H}$ eine Folge erhalten, die bezüglich der Norm in $\mathbb{H}$ divergiert oder nicht. Den Abschluß der “divergenten Punkte” bezeichnen wir wieder als Juliamenge. Diese Punkte liegen jedoch in einem 4-dimensionalen Vektorraum. Um sie darstellen zu können schneiden wir sie mit einem 3-dimensionalen Hyperraum. Es ist zu erwarten, daß sich interessante graphische Resultate beim Verschieben dieses Hyperraums ergeben, was sich durch Experimente bestätigt.

Im Körper der komplexen Zahlen existiert eine fundierte mathematische Theorie, die die Struktur von Juliamengen beschreibt [1]. Sie basiert vor allem darauf, daß die hierbei untersuchten Funktionen in $\mathbb{C}$ analytisch sind und deswegen viele Resultate aus der Funktionentheorie angewendet werden können. In $\mathbb{H}$ ist der Begriff der Differenzierbarkeit zwar definierbar, allerdings ist die Klasse der differenzierbaren Funktionen zu klein, als daß man erwarten könnte die Resultate von $\mathbb{C}$ nach $\mathbb{H}$ zu übertragen (z.B. sind Polynome i.A. nicht differenzierbar).

Der Mangel an mathematisch genauer Beschreibung dieser Mengen macht den Einsatz von Computergraphik um so interessanter.

2.3. **Rotationen mit Quaternionen.** Die Maussteuerung des Viewers (siehe Abschnitt 4), mit der man die Szene drehen kann, haben wir mit Hilfe von Quaternionen implementiert. Dabei wird der Positionsvektor um momentane Achsen durch den Ursprung, welche parallel zur “hoch”-Richtung bzw. zur “Seit”-Richtung liegen, oder auch die “hoch”-Richtung um die Blickrichtung (“rollen”) in kleinen Schritten rotiert (siehe Abbildung 1). Im Grunde galt es also, die Aufgabe zu bewältigen, einen beliebigen 3 dimensionalen Vektor um eine beliebe Achse zu drehen.

Zunächst war unser Ansatz, Eulermatrizen zu verwenden. Dies stellte sich jedoch als eher kompliziert heraus. Ironischerweise waren Quaternionen, eigentlich Gegenstand dieses Viewerprogramms, dann die Lösung des Problems.

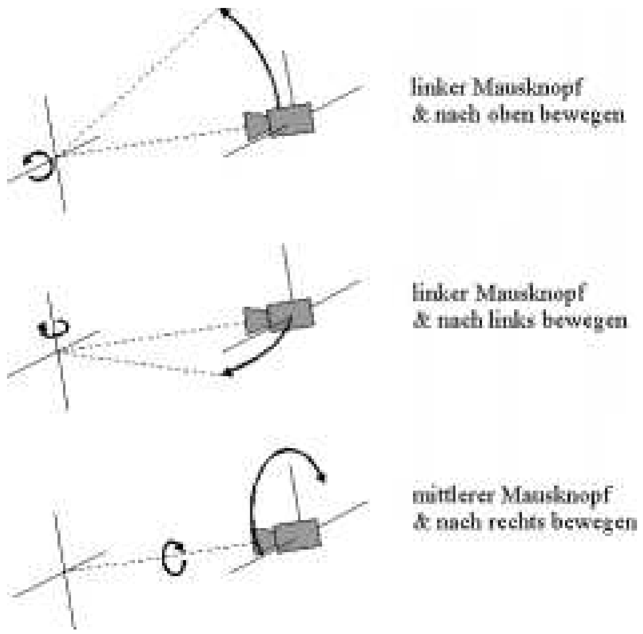

ABBILDUNG 1. Rotationen im Viewer

Die Realisierung mit Hilfe von Quaternionenrotation erwies sich als recht einfach. Dabei nutzten wir unsere Implentierung von Quaternionen und ihrer Rechenoperationen in der Klasse `QuatMath`. Wir verzichten hier auf eine genaue mathematische Erklärung der Rotationsdarstellung durch Quaternionen und geben lediglich den (einfachen) Algorithmus an. Eine ausführlichere Beschreibung findet man z.B. bei *planetmath.org* [4].

Wir wollen einen Vektor $x = (x_0, x_1, x_2)$ um die Achse $a = (a_0, a_1, a_2)$ um den Winkel θ drehen. Dazu betten wir den Vektor in die Quaternionenalgebra ein, indem wir setzen

$$v = 0 + x_0 i + x_1 j + x_2 k$$

Desweiteren bilden wir das "Rotationsquaternion"

$$q = \cos(\theta/2) + \sin(\theta/2)(a_0 i + a_1 j + a_2 k)$$

Mit der Konjugation q^* von q (siehe oben) bilden wir

$$v' = qvq^* = 0 + x'_0 i + x'_1 j + x'_2 k$$

Dann ist $x' = (x'_0, x'_1, x'_2)$ der gesuchte gedrehte Vektor.

3. ALGORITHMEN

3.1. **Problematik.** Wie oben beschrieben, entsteht eine Juliamenge J_f durch Angabe eines Polynoms $f(X)$ und ist dadurch eindeutig bestimmt. Die Menge der unter f^n divergenten Punkte wird oft als *Runaway Set* bezeichnet, das Komplement von J_f heißt *Prisoner Set*. Runaway bzw. Prisoner Punkte seien im folgenden als OUT bzw. IN Punkte bezeichnet. Um J_f im Quaternionenraum $\mathbb{H}$ darzustellen, betrachten wir den Schnitt von J_f mit einem dreidimensionalen Untervektorraum von $\mathbb{H}$. Wir wollen die Grenzfläche zwischen IN und OUT Punkten in diesem Schnitt polygonisieren.

Der dazu verwendete Algorithmus basiert auf dem Implicit Surface Polygonization Algorithmus von Jules Bloomenthal[2]. Es wurden allerdings einige Änderungen vorgenommen, um ihn auf das Darstellen der fraktalen Oberfläche anzupassen.

Es wird zunächst der 3D-Schnitt vorgenommen. Hierzu haben wir vorerst die Hyperebene $\mathbb{E} = \{q \in \mathbb{H} \mid q_3 = 0\}$ gewählt. In $\mathbb{E}$ wird ein 3-dimensionales Intervall bestimmt, in dem das Fraktal dargestellt werden soll. Dieses erhält der Algorithmus als Parameter. Das Intervall wird nun iterativ mit Würfeln einer festzulegenden Kantenlänge ausgefüllt. Es wird berechnet, welche Punkte an den Würfeleckpunkten IN bzw. OUT Punkte sind. Dabei wird ein Konvergenz Cache eingerichtet um redundanten Rechenaufwandt zu sparen. Ein Würfel schneidet die Fraktaloberfläche genau dann, wenn sowohl Eckpunkte IN sind, als auch Eckpunkte OUT sind. Ist dies der Fall, so wird für diesen Würfel der folgende Polygonisierungs-Algorithmus gestartet.

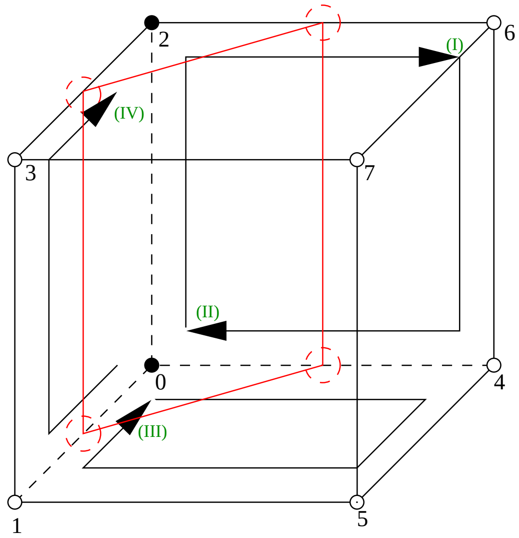

ABBILDUNG 2. Algorithmus zur Polygonisierung

3.2. **Cube Polygonize Algorithm.** Die Würfelecken werden wie in Abbildung 2 durchnumeriert. Wir halten folgende Elemente aus der Permutationsgruppe der Würfelecken fest:

$$\sigma_1 = (0264) \circ (1573)$$

$$\sigma_2 = (0132) \circ (4675)$$

$$\sigma_3 = (0451) \circ (2376)$$

In römischen Ziffern habe ich Vermerke auf Abbildung 2 eingefügt, die zum besseren Verständnis des Algorithmus beitragen sollen. Hier sind exemplarisch Punkte 0 und 2 IN und alle anderen OUT.

Nach Vorraussetzung gibt es mindestens einen IN Eckpunkt und mindestens einen OUT Eckpunkt im Würfel. Es wird wie folgt ein Polynom konstruiert, welches die Fraktalhülle im Würfel approximiert.
Wir beginnen an einem IN Punkt i und wenden σ_* auf diesen solange an, bis wir an einem OUT Punkt sind. Wir betrachten also $\{\sigma_1(i),\ \sigma_1^2(i),\ \sigma_1^3(i),\ \sigma_2(i),\ \sigma_2^2(i),\ \ldots\}$. Falls dies nicht klappt (genau dann, wenn der i diagonal gegenüberliegende Punkt der einzige OUT Punkt ist), ersetzen wir i durch $\sigma_1(i)$ und wiederholen das Verfahren. Dies liefert nun auf jeden Fall einen OUT Punkt. $\Rightarrow (I)$
Das zuletzt benutzte σ_l wird jetzt iterativ auf diesen Punkt angewendet, was uns in jedem Falle einen IN Punkt j und den vorher durchlaufenen OUT Punkt $k = \sigma_l^{-1}(j)$ liefert (Wir sind bei einem IN Punkt gestartet und haben σ_l auf diesen angewandt, das heißt schlimmstenfalls landen wir wieder bei diesem Ausgangspunkt und können daher sicher sein, einen IN Punkt zu erwischen). $\Rightarrow (II)$
BEGIN:
Wir wissen nun, daß zwischen j und k ein Wechsel von IN nach OUT stattgefunden hat. Das heißt, die Kante zwischen diesen Punkten schneidet die Fraktalhülle. Also setzen wir einen Vertex unseres Polygons in die Mitte dieser Kante. Nun wählen wir wieder dasjenige σ_l aus, welches j zurück auf k abbildet und wenden es iterativ an, bis wir wieder bei einem IN Punkt j' landen. $\Rightarrow (III)$
Wieder merken wir uns auch den zuvor durchlaufenen OUT Punkt k' und gehen mit $j <= j'$ sowie $k <= k'$ zurück zu BEGIN.

Dieses Verfahren wiederholen wir, bis wir versuchen einen Vertex ins Polygon aufzunehmen, den wir schon gespeichert haben. Dann haben wir innerhalb des Würfels ein Polygon bestimmt, dessen Eckpunkte genau auf denjenigen Kanten liegen, deren Eckpunkte von IN nach OUT wechseln. Dies liefert eine gute lokale Approximation für die Oberfläche des Fraktals. Die Verwendung der Permutationen vereinfacht den Algorithmus deutlich und stellt sicher, daß alle Schnittkanten erfaßt werden.

Nachträglich wird eine Normale berechnet, indem das Kreuzprodukt aus zwei Polygonkanten gebildet wird. Diese Methode birgt jedoch Fehler, falls die Polygoneckpunkte nicht in einer Ebene liegen und ist daher noch verbesserungsfähig.

Zu Bemerken ist, daß das Polygon im Würfel durch die IN-OUT-Konstellation der 8 Eckpunkte eindeutig bestimmt ist. Somit ergeben sich $2^8 - 2 = 254$ verschiedene Polygone, welche jeweils auch die selbe Normale haben. Diese Polygone sind vollkommen unabhängig von den Fraktalparametern und werden daher bei der Initialisierung des Programmes in einen sogenannten Polygon Cache geschrieben. Während des Algorithmus wird mittels einer 8 Bit langen Bitmap (8 Eckpunkte) direkt auf diesen Polygon Cache zugegriffen, was einiges an Performance ausmacht.

Alle Polygone werden in einen Vektor gespeichert. Nachdem das 3D-Intervall vollständig durchlaufen wurde, müssen die Normalen gemittelt werden. Für jeden

Punkt, der in einem Polygon vorkommt, werden die Normalen von allen Polygonen in denen dieser Punkt vorkommt gemittelt.

Die so erhaltenen Polygone werden dann in eine GL display list geschrieben und sind bereit, dargestellt zu werden.

3.3. **Alternativen und Ausblick.** Während des Praktikums haben wir auch viele Ideen zu anderen Algorithmen entwickelt, die wir bis jetzt nur unvollständig implementieren konnten. Um das oben erwähnte Normalenproblem zu beheben, haben wir einen *Tetra Polygonize Algorithm* implementiert, der jedoch zu einer erheblich "kantigeren" Darstellung führte und deswegen zunächst beiseite gelegt wurde. Es wäre auch eine Kombination beider Algorithmen denkbar, um die jeweiligen Vorteile ausnutzen zu können.

Den *Cube Polygonize Algorithm* haben wir in den letzten Wochen noch grundlegend optimieren können, so daß er jetzt eine sehr gute Performance aufzeigt. Eine weitere Optimierungsmöglichkeit wäre, einen *Surface Tracking Algorithm* zu implementieren, der mit einem Würfel auf der Fraktaloberfläche beginnt und von dort aus nur Würfel berücksichtigt, die auch wirklich auf der Oberfläche liegen. Da man bei quadratischem $f(X) = X^2 + C$ sofort überprüfen kann, ob die Fraktalhülle zusammenhängend ist (genau dann, wenn $f^n(0)$ konvergiert), würde die Methode in diesen Fällen enorme Rechenzeit sparen.

Es gibt Aussagen[3], daß die Juliamengen von quadratischen Funktionen aus Rotation einer komplexen Juliamenge um eine bestimmte Achse entstehen. Auch dies könnte enorme Recheneinsparungen liefern.

4. QuatFract - Das Programm

4.1. **Methoden.** Zunächst eine kleine Zusammenstellung der von uns benutzten Tools und Bibliotheken:
Als Programmiersprache wählten wir *C++*[5], da wir Objektorientierung benutzen wollten (aus Gründen die in Abschnitt 3.3 erläutert werden). Wir waren beide schon etwas mit der Grafikbibliothek *Open GL*[6] vertraut und verwendeten diese deshalb zur grafischen Darstellung. Als Widget Bibliothek bot sich *GTK++*[7] bzw. dessen C++ wrapper *gtkmm*[8] an, da die Kompatibilität mit Open GL schon durch die Erweiterung *glareamm*[9] gesichert war. Ausserdem kamen C++ Bibliotheken für Quaternionen- und Permutationsmathematik zur Verwendung, die einer von uns schon im Vorraus implementiert hatte. Für das Rendern einer Kamerafahrt wurde der Raytracer *povray*[10] benutzt.

Für das Versionsmanagement benutzten wir *CVS*[11] und zur automatischen Klassendokumentation *doc++*[12]. Ausserdem diente eine kleine Website[13] mit einem mit *PHP4* [14] geschriebenem Projekttagebuch zur Dokumentation des Praktikumverlaufs. Dort sind auch Screenshots, CVS log history, CVS snapshots, aktuelle releases und Dokumentation vorhanden. Da das Programm noch weiterentwickelt werden soll, wird diese Website auch weiterhin betrieben werden.

Der Code ist erhältlich unter den Bedingungen der *GNU General Public License*[15].

4.2. **Kurzbeschreibung.** Unser Programm QuatFract wird bedient über ein Hauptfenster. Den größten Teil des Fensters füllt der OpenGL Display aus, in dem unter anderem das Fraktal dargestellt werden kann. Darüber befindet sich eine "toolbar", sprich eine Werkzeugleiste, mit mehreren Knöpfen. Die meisten Knöpfe rufen Unterfenster auf, einer startet das Berechnen und Rendern der Fraktalhülle mit den derzeitigen Einstellungen. Wird der Mauszeiger über einen der Knöpfe gehalten, erscheint ein "tooltip", der über dessen Funktionsweise Auskunft gibt. Mit Hilfe der Maus kann normalerweise die dargestellte Szene rotiert werden, manche Unterfenster können aber alternative Maussteuerungen aktivieren. Im Einzelnen gibt es augenblicklich folgende Unterfenster:

- *Viewer Tools*
  - `Das Kamera-Fenster.`
    Über dieses Fenster können die aktuellen Daten der Kamera, von der aus der Benutzer die Szene betrachtet, beobachtet oder direkt eingegeben werden, z.B. die Positon oder der Blickpunkt. Außerdem kann eine alternative "Move"-Maussteuerung ausgewählt werden. Desweiteren läßt sich hier auch die Position der Lichtquelle ändern.
- *Fractal Tools*
  - `Das WaypointEditor-Fenster.`
    Hier können Wegpunkte eingefügt und editiert werden. Die Wegpunkte definieren die Kamerafahrt, welche mit Hilfe des Povray Output Fensters an Povray übergeben und gerendert werden kann.
  - `Das PovrayOutput-Fenster.`
    Die mit dem Waypoint Editor erstellte Liste von Wegpunkten wird mit Hilfe dieses Fensters in povray template files eingefügt. Die ausgegebenen Dateien definieren povray Szenen, welche die gewünschte Kamerafahrt als Bildsequenz rendern. Hierbei wird ein povray-interner Clocking-Mechanismus benutzt, um die verschiedenen Wegpunkte nacheinander abzulaufen.
  - `Der Render-Knopf.`
    Wird der Knopf gedrückt, so wird die Fraktalhülle mit den aktuellen Einstellung berechnet und gerendert. Dieser Vorgang kann je nach Größe des zu untersuchenden Gebiets, Rasterdichte und Iterationstiefe kürzer oder länger dauern.
  - `Das FractalParameters-Fenster.`
    Hier können erstens die "Rendering"-Optionen eingestellt werden, d.h. der zu untersuchende Ausschnitt des Raumes, definiert über die "Boundary Box", und die Rastergenauigkeit. Die Größe und Position der Boundary Box kann auch mit Hilfe der Maus verändert werden, wenn die entsprechenden Knöpfe im Fenster aktiviert werden. Nur innerhalb

der Box wird beim Drücken des Render-Knopfes das Fractal berechnet und gerendert. Zweitens können Einstellungen an der Fraktalberechnung vorgenommen werden, z.B.Iterationstiefe und die Konstante der quadratischen Iterationsfunktion. Die Funktion selbst kann noch nicht verändert werden.

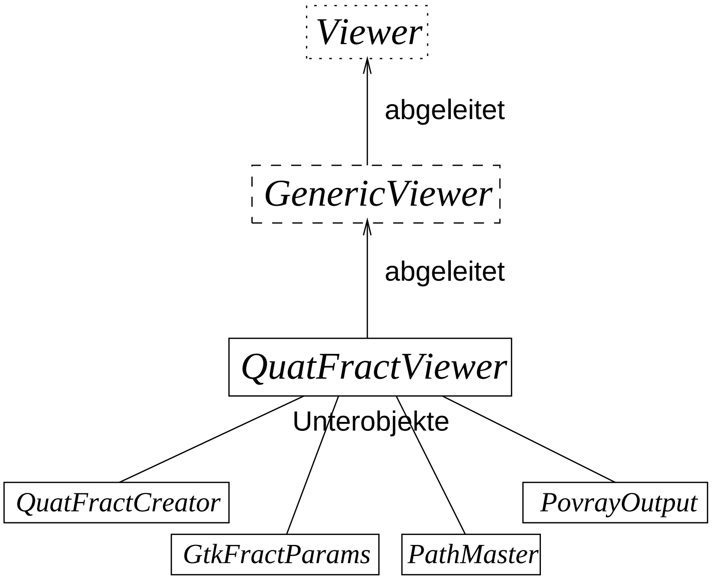

Abbildung 3. Klassenhierarchie

4.3. **Programmstruktur.**

4.3.1. *Klassenhierarchie: Viewer & Co.* Für QuatFract brauchten wir zunächst einen Viewer, der OpenGL Inhalte darstellen konnte. Natürlich wäre ein solches Programm auch nützlich für viele andere Anwendungen. Dies im Gedächtnis entwarfen wir zunächst die `Viewer` Klasse, bestehend aus einem einzigen Gtk Fenster und darin einem `Gtk::GLArea` widget, welches den OpenGL Inhalt rendert. Der `Viewer` verfügt außerdem schon über eine ganz ordentliche Maussteuerung, mit der man die Szene rotieren kann. Diese Rotation haben wir mit Hilfe von Quaternionen-Operationen realisiert, siehe dazu Abschnitt 2.3. Einige wichtige Methoden in `Viewer` sind virtuell, darunter Methoden zum Initialisieren und Zeichnen eines eigenen OpenGL Inhalts, und mehrere Methoden, die aufkommende Mausereignisse "handeln", also das Klicken und Bewegen der Maus verarbeiten. In Letzteren ist die `Viewer` Maussteuerung implementiert. Durch Überschreiben dieser Methoden kann man eine alternative Maussteuerungen realisieren (und je nachdem auch die ursprüngliche Maussteurung nutzen, indem die `Viewer` Methoden aufgerufen werden). Das Code Beispiel im Anhang demonstriert eine mögliche Verwendung der `Viewer` Klasse, um einen eigenen spezifischen Viewer abzuleiten.

Außerdem gibt es öffentliche Gtk Container im Hauptfenster, welche dann von abgeleiteten Klassen mit zusätzlichen widgets gefüllt werden können.

Ein Beispiel für den `Viewer` als Basisklasse ist auch unser `GenericViewer`. Dieser kommt als nächstes in der Vererbungshierarchie. Er ist gedacht als Viewer mit ein paar zusätzlichen Funktionen, welche aber immernoch allgemein gehalten sind, so daß der `GenericViewer` für viele andere Programme die Basis darstellen kann. Der `GenericViewer` verfügt über eine toolbar, die oberhalb vom GL Display platziert ist. Im Augenblick befindet sich darin nur ein einziger Knopf, der das sogenannte Camera Fenster aufruft (Beschreibung s.o.). Hier kann eine alternative "move"-Maussteurung aktiviert werden. Der neue Modus ist in den virtuellen Maus Handler Methoden implementiert.

4.3.2. *Der QuatFractViewer.* Last but not least kommt der `QuatFractViewer`, abgeleitet vom `GenericViewer`. Er stellt den Kern unseres Programms dar. In `main()` wird nur ein `QuatFractViewer` instanziert und der Gtk event loop gestartet. Der `QuatFractViewer` fügt neue tools ("Fractal Tools") in die Werkzeugleiste ein und behandelt aufkommende Knopfdruck-events. In den meisten Fällen soll ein neues Unterfenster gestartet werden. Alle diese Unterfenster sind implementiert in eigenen Klassen, von denen der `QuatFractViewer` je eine Instanz als member besitzt. Dazu kommen weitere Unterobjekte, die bestimmte Aufgabengebiete übernehmen. Der `QuatFractViewer` ist also zentrale Bedinungsoberfläche, Viewer des GL Inhalts und Manager der verschiedenen Unterprogramme. Manche dieser Unterfenster benötigen die Kontrolle über die Maus, z.B. wenn mit Hilfe des Fractal Parameters Fensters die Boundary Box verschoben werden soll. Dann gibt der `QuatFractViewer` die Mausereignisse an das Unterfenster weiter. Oftmals müssen diese Unterobjekte also mit ihrem Parent, dem `QuatFractViewer`, kommunizieren, z.B. um Mauskontrolle zu erbitten oder anzuzeigen, daß die Darstellung des OpenGL Inhalts aktualisiert werden muß. Deshalb bietet es sich an, an dieser Stelle des Berichts kurz zu erläutern, was schon öfters erwähnt wurde, und wovon wir intensiv Gebrauch gemacht haben: Die Kommunikation mit Signalen in einem Gtk Programm.

4.3.3. *Signale und deren Behandlung.* Immer wenn in einem Gtk Programm z.B. ein Knopf gedrückt wird, erzeugt dieser ein entsprechendes event/signal. Das dem Knopf übergordnete Objekt, z.B. das Fenster, indem sich der Knopf befindet, kann diesem event einen Handler zuweisen, indem er es "verbindet", mit einem Aufruf, der in etwa so aussieht:

```
button.clicked.connect(...this, &MyClass::my_handler...)
```

In Gtkmm kann es sich beim Handler um Funktionen oder wie hier um Methoden von Instanzen handeln. Nun ist es auch möglich, Klassen eigene Signale als Unterobjekte zu geben. Diese Signale können auch Argumente erhalten. Das Signal wird wie eine Methode aufgerufen. Übergeordnete Klassen können entscheiden, welche Handler sie an diese Signale binden möchten.
Beispiel: In QuatFract kann der Benutzer in manchen Unterfenstern den darzustellenden OpenGL Inhalt ändern, z.B. indem er einen neuen Waypoint hinzufügt.

Das entsprechende Unterfenster sendet dann das (von uns so genannte) “redraw”-Signal. Dieses Signal wird vom übergeordneten `QuatFractViewer` einfach mit der Zeichenmethode `drawgl()` verbunden, und zwar so:

```
pathmaster.redraw.connect(slot(this, &QuatFractViewer::drawgl))
```

Diese Kommunikation mit Hilfe von Signalen hat sich in unserem Programm als sehr praktisch erwiesen. Da Signale auch Rückgabewerte haben können, haben wir sie auch dazu verwendet, Informationen vom `QuatFractViewer` abzufragen: Manche Unterfenster benötigen manchmal die Daten der Kamera, welche im `QuatFractViewer` (bzw. in der Elterklasse `Viewer`) in einem Unterobjekt `camera` gespeichert sind. Sie senden dann ein “camara_stats_input”-Signal, das vom `QuatFractViewer` an entsprechende Methoden des camera-Unterobjekts verbunden wird, und dann die Daten zurückgibt.

4.3.4. *Die wichtigsten Unterklassen des QuatFractViewers.* Wie oben beschrieben sind die wichtigsten Teile von QuatFract als Unterobjekte im `QuatFractViewer` integriert. Dabei handelt es sich konkret um folgende Klassen, die nun kurz erläutert werden sollen:

- `QuatFractCreator`
  Der `QuatFractCreator` ist die Einzige dieser Unterklassen, die nicht ein Unterfenster des Programms darstellt. Hier werden alle Informationen über die darzustellende Fraktalhülle gesammelt. Die Algorithmen zur Berechnung des Fraktals und für das Rendern der Polygonhülle befinden sich alle hier, meist ausgelagert in eigene Klassen. Der `QuatFractViewer` mit seinen Algorithmen etc. existiert insbesondere unabhängig von der Benutzeroberfläche, ist also Gtk-frei. Dies erlaubt, eventuellen Portierungswünschen zu anderen Widget-Systemen nachzugehen.

- `GtkFractParams`
  In diesem Fenster sollen Einstellungen am Berechnen und Rendern vorgenommen werden. Die Klasse ist, wie auch der `Viewer`, von `Gtk::Window` abgeleitet. Im Fenster befindet sich ein `Gtk::Notebook`, mit zwei Karteikarten, von denen eine die Renderoptionen und die andere die Berechnungsfunktionen beinhaltet. Diese Widgets befinden sich nochmals in zwei eigenen Klassen, von denen `GtkFractParams` je eine Instanz als Unterobjekt besitzt. Anders als beim Hauptfenster wird das FractParams Fenster nicht wirklich zerstört,wenn es über die toolbar oder direkt im Fenster geschlossen wird. Stattdessen haben wir entsprechende virtuelle Methoden von `Gtk:Window` überschrieben, um das Fenster nur zu “hiden”, sprich unsichtbar zu machen. Das Fenster bleibt stets Unterobjekt im `QuatFractViewer`.

- `PathMaster`
  Der `PathMaster` verwaltet den sogenannten “path”, sprich die Wegpunkte für den Kameraflug. Die Klasse beinhaltet eine Unterklasse, die das `WaypointEditor` Fenster darstellt. Anders als bei `GtkFractParams` und

`PovrayOutput` ist der PathMaster also nicht selbst von `Gtk:Window` abgeleitet; außerdem haben wir hier versuchsweise nicht das übliche showhide implementiert, sprich das Verbergen des Fensters beim Schließen (siehe `GtkFractParams`). Stattdessen wird das Editor Fenster tatsächlich zerstört, wenn es geschlossen wird, und neu erzeugt, wenn es über den Knopf in der toolbar gestartet wird. Demnach besitzt der `PathMaster` auch nur einen Zeiger auf ein Objekt `WaypointEditor`, das er bei Bedarf neu erzeugt oder löscht. Sinn und Zweck war, vielleicht Speicherplatz zu sparen und nicht unzählige unsichtbare Fenster im Hintergrund laufen zu haben. Tatsächlich ergaben sich jedoch Probleme (man beachte z.B.: das Editor Fenster bleibt nicht wie andere Unterfenster im Vordergrund). Wahrscheinlich werden wir deswegen wieder showhide benutzen.

- `PovrayOutput`
  Diese Klasse ist wiederum von `Gtk::Window` abgeleitet. Hier kann der path in geeignete Files übersetzt werden, die von povray gerendert werden. Wie in `GtkFractParams` gibt es im Fenster mehrere Karteikarten, die der Übersicht halber in eigenen Klassen implementiert sind. Die Ausgabe funktioniert mit Hilfe von Dateischablonen, die mit speziellen Ausdrücken der Gestalt "!EXPRESSION!" versehen sind. Diese Ausdrücke erkennt unser parser und kann sie mit vom Benutzer gewünschten, über die Gtk GUI wählbaren, Variablen ersetzen. Dies erlaubt die Festlegung aller für die povray Szene relevanten Einstellungen (z.B. Fraktalparameter, Sequenzlänge, Farbe, Licht, Ausgabedatei, ...). Für die Kamerafahrt wird von `PovrayOutput` ein Streckenzug durch alle Wegpunkte (waypoints) berechnet und unter Benutzung des povray Clocking features in der Ausgabedatei gespeichert. Es werden eine *xample.pov* und eine *xample.ini* Datei erstellt, welche mit dem Befehl *povray xample.ini* eine gerenderte Bildsequenz erzeugen.

4.3.5. *Zu den anderen Programmteilen.* Desweiteren haben wir zahlreiche mehrfach verwendete widgets in eigene Klassen gepackt, z.B. Texteinträge für 3 dimensionale Koordinaten. Auch an anderer Stelle nutzten wir die Möglichkeit zur Vererbung von Basisklassen. Die Klasse mit dem klangvollen Namen `SpaceEntity` z.B. beschreibt ein Objekt im 3-dimensionalen Raum mit einer Orientierung (Position, Blick- und "Oben"-richtung). Die `camera` in `Viewer` ist eine Instanz dieser Klasse. Die `Waypoint` Klasse wiederum beinhalted alle Daten, die für die Wegpunkte benöigt werden; zu Position und Orientierung kommt eine vierte Koordinate (4D Quaternionenraum!) und ein Zeitwert, der bestimmen soll, wann die Kamera während ihres Fluges bei diesem Wegpunkt ankommt. Die `Waypoint` Klasse ist also von `SpaceEntity` abgeleitet.

Eine Zusammenstellung aller verwendeten Klassen mit Dokumentation ist in Form einer Referenz zusammengestellt und auf unserer Website[13] zu finden.

Literatur

[1] Kenneth J. Falconer, *Fraktale Geometrie*. Spektrum Akademischer Verlag (1993)
[2] Jules Bloomenthal, *An Implicit Surface Polygonizer*. Graphics Gems IV (1994)
[3] A. A. Bogush, A. Z. Gazizov, Yu. A. Kurochkin and V. T. Stosui, *On Symmetry Properties of Quaternionic Analogs of Julia Sets.* (2001)
[4] *http://planetmath.org/encyclopedia/SomethingRelatedToQuaternions.html*
[5] *http://www.research.att.com/ bs/C++.html*
[6] *http://www.opengl.org*
[7] *http://www.gtk.org*
[8] *http://www.gtkmm.org*
[9] *http://www.ece.ucdavis.edu/ kenelson/gtkglareamm/index.html*
[10] *http://www.povray.org*
[11] *http://www.cvs.org*
[12] *http://www.zib.de/Visual/software/doc++/*
[13] *http://www.rzuser.uni-heidelberg.de/˜tdyckerh/*
[14] *http://www.php.net/*
[15] *http://www.gnu.org/copyleft/gpl.html*

Beispiel Code zur Ableitung von Viewer

```
#include "Viewer.h"


class ViewerPlus: public Viewer
{

  /* these are the methods which handle the various mouse events,
     namely motion, button press and button release */

  virtual gint gl_motion_notify_handler_imp(GdkEventMotion* event);
  virtual gint gl_button_press_handler_imp(GdkEventButton* event);
  virtual gint gl_button_release_handler_imp(GdkEventButton* event);

  /* these methods are used to initialize and draw our GL content */

  virtual gint init_gl_context_imp();
  virtual gint draw_gl_context_imp();

  /* you have to call Viewer::drawgl() when you want your content to
     be displayed!! */


  /* say we want to realize a second mouse control; we could use a
     variable to tell which mode is currently active */

  int mouse_mode;


public:

  /* our constructor takes a string as argument which will label our
     window */
  ViewerPlus(char* label);
};


/* the label is passed to the constructor of the Viewer */
ViewerPlus::ViewerPlus(char* label): Viewer(label)
{
  mouse_mode=2;
}



gint GenericViewer::gl_motion_notify_handler_imp(GdkEventMotion* event)
{
  /* here we look which mouse mode is active and call our own mouse
     control or the original one of the Viewer */

  /* old mode:the Viewer methods are called */
  if (mouse_mode==2) Viewer::gl_motion_notify_handler_imp(event);
```

```
  /* new mode */
  if (mouse_mode==1)
    {
      /* implement new mode here */

      /* if we need to redraw the scene due to a change of the content
         or the camera stats we call this: */

      Viewer::drawgl();
    }



}


gint GenericViewer::gl_button_press_handler_imp(GdkEventButton* event)
{
  /* old mode:the Viewer methods are called */
  if (mouse_mode==2) Viewer::gl_button_press_handler_imp(event);

  /* new mode */
  if(mouse_mode==1)
    {
      /* implement new mode here */
    }
}

gint GenericViewer::gl_button_release_handler_imp(GdkEventButton* event)
{
  /* old mode:the Viewer methods are called */
  if (mouse_mode==2) Viewer::gl_button_release_handler_imp(event);

  /* new mode */
  if (mouse_mode==1)
    {
      /* implement new mode here */
    }
}


/* finally these are our OpenGL methods*/

gint QuatFractViewer::init_gl_context_imp()
{
  /* here would be a nice place to initialize some display lists for
     example */
}


gint QuatFractViewer::draw_gl_context_imp()
{
```

```
  /* here we would make our GL drawing calls */
}
```