UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Erzeugung eines dynamischen, zufälligen und endlosen Terrains

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Pascal Dietz

Betreuer: Dipl.-Inform. Oliver Abert
Arbeitsgruppe Computergrafik an der Universität Koblenz

Koblenz, im Februar 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Studienarbeit war es, Erfahrungen in der Grafik- und Spieleprogrammierung zu sammeln. Als Grundidee kam dabei die Erstellung eines 3-dimensionalen Terrains auf. Solche Terrains werden heutzutage nicht nur in der Spielebranche eingesetzt, wo sie in beinahe jedem Genre vertreten sind, sondern auch z.B. in der Geologie zur Erstellung von Simulationen von Plattentektonik. Die simple Erstellung eines 3-dimensionalen Terrains wäre für eine Studienarbeit jedoch zu trivial, daher sollte das Terrain spezielle Anforderungen erfüllen.
Zum einen sollte das Terrain dynamisch erzeugt werden, d.h. der Benutzer des Programms hat Einfluss darauf, wie sich das Terrain entwickelt. Dies sollte vorzugsweise spielerisch eingebracht werden.
Zum anderen sollte das Terrain zufällig generiert werden. Dies bedeutet, dass keine vormodellierte Landschaft genutzt, sondern jede Erhebung/-Vertiefung des Terrains mittels Zufallsfaktoren erzeugt werden sollte.
Zusätzlich sollte das Terrain endlos erzeugt werden. Bei einer Bewegung über das Terrain sollte also niemals ein Ende erreicht werden. Also auch keine Kreistrecke, sondern ein wirklich endloses und stets anders aussehendes Terrain.
Desweiteren sollte es dem Benutzer møglich sein, ein Fluggerät über das Terrain zu steuern. Dies gab dann auch die Chance, aus der oben genannten dynamischen Anforderung ein spielerisches Element zu machen, indem der Benutzer das Terrain durch Einsammeln von sogenannten *TerraformItem*s beeinflussen kann. Die Steuerung eines Fluggerätes spielt auch für die geforderte Endlosigkeit des Terrains eine wichtige Rolle, da diese ohne eine Möglichkeit der Fortbewegung gar nicht nachprüfbar wäre. Das Problem mit der Endlosigkeit ist dabei, dass kein System endlosen Speicher zur Verfügung hat um das Terrain komplett zu speichern und dem Benutzer somit die Option zu bieten, die gleiche Strecke zurückzufliegen. Eine Lösung für diese Problematik wäre bei einer Kehrtwende das Terrain auch rückwärts wieder neu zu generieren. Der Einfachheit halber sollte stattdessen ein komplette Kehrtwende einfach nicht zugelassen werden.
Eine Kollisionserkennung musste dann natürlich auch implementiert werden. Zum einen weil das Fluggerät ja nicht einfach wie ein Geist durch das Terrain hindurchgleiten sollte, zum anderen muss das Programm ja irgendwie das Einsammeln der oben angesprochenen *TerraformItem*-Objekte registrieren können. Weitere Objekte wie Bäume oder Felsen sollten das Terrain optisch aufwerten.
Zu guter Letzt sollte noch eine simple Benutzeroberfläche erstellt werden, um dem Benutzer diverse Bedienelemente und Rückmeldungen zu bieten. Damit sollte es z.B. auch möglich sein dass Terrain direkt zu verändern.

2 Grundlagen

2.1 OpenGL

Die *Open Graphics Library* (kurz: *OpenGL*, siehe [OpenGL]) ist eine der meistgenutzten 3D APIs (*application programming interface*, zu deutsch 'Schnittstelle zur Anwendungsprogrammierung') der heutigen Zeit. Eine API ist eine Software, die es anderen Programmen u.a. ermöglicht, auf verschiedene Systemkomponenten wie z.B. die Grafikkarte oder auf Datenbanken zuzugreifen. *OpenGL* stellt Softwareentwicklern Befehle zur Verfügung, mit denen sie die Grafikkarte ansprechen können um komplexe 3d-Grafiken zu erstellen.
OpenGL ist eine Zustandsmaschine. D.h. wenn z.B. eine Farbe zum Zeichnen festgelegt wurde, bleibt diese erhalten bis sie wieder geändert wird. Dies hat den Vorteil, dass Angaben wie die Farbgebung oder die Transparenz nicht für alle Vertices die gezeichnet werden sollen erneut angegeben werden müssen, wenn diese die gleichen Eigenschaften erhalten sollen. Das macht den Umgang mit *OpenGL* an vielen Stellen sehr komfortabel.
Für die weite Verbreitung von *OpenGL* ist wohl auch die weitgehende Plattformunabhängigkeit verantwortlich, d.h. dass es auf verschiedensten Computersystemen mit Unterschieden in Architektur, Prozessor, Compiler oder Betriebssystem einsetzbar ist.
Ebenso ist die Erweiterbarkeit von *OpenGL* ein wichtiger Punkt. Anbieter (i.d.R. Grafikkartenhersteller) können die Zustandsmaschine von *OpenGL* um eigene Zustände erweitern um so dessen Funktionalitäten auszubauen.
Die Tatsache, dass man keine Lizenz für den Gebrauch von *OpenGL* erwerben muss (Grafikkartenhersteller ausgenommen), trägt sicherlich auch dazu bei, dass dies ein so beliebtes Werkzeug zur Grafikherstellung geworden ist.
Entwickelt wurde *OpenGL* 1992 von Silicon Graphics, seit Juli 2000 wird es jedoch von der Khronos Group verwaltet, einer Vereinigung mehrerer Unternehmen, die sich für die Erstellung und Verwaltung offener Standards im Multimediabereich auf einer Vielzahl von Plattformen und Geräten einsetzt. Aktuell gibt es *OpenGL* in der Version 2.1.

2.2 Qt

Qt (siehe [QT]) ist eine plattformunabhängige Klassenbibliothek zur Programmierung mit *OpenGL* die von der Firma Trolltech entwickelt wird. Es verwendet eine erweiterte Version von der Programmiersprache C++, wurde mittlerweile jedoch auch für viele andere Sprachen implementiert. Diese werden jedoch nicht von Trolltech betreut.
Mit *Qt* können *OpenGL*-Kontexte erzeugt und grafische Benutzeroberflächen

erstellt werden. Es bietet viele Erweiterungen fü C++ an, unter anderem auch das SIGNAL-SLOT-Konzept, welches an späterer Stelle erläutert wird. Dies macht *Qt* interessanter als Alternativen wie z.B. das recht bekannte GLUT (*OpenGL Utility Toolkit*, welches deutlich weniger Funktionalitäten liefert und eher als Einstieg für Programmierer ohne *OpenGL* Erfahrung interessant ist als für professionelle Entwickler.

2.3 C++

In dieser Studienarbeit wurde die Programmiersprache C++ verwendet. C++ unterstützt mehrere Programmierparadigmen wie objektorientierte, prozedurale und generische Programmierung und wird vor allem in der Systemprogrammierung genutzt. Aus der Anwendungsprogrammierung wurde C++ weitgehend von anderen Sprachen wie Java und C# verdrängt. C++ wurde für diese Studienarbeit aufgrund studiumsbedingter Vorkenntnisse und Programmiererfahrung gewählt. Entwickelt wurde unter WindowsXP mit dem Microsoft Visual Studio .NET 2003. Visual Studio ist eine integrierte Entwicklungsumgebung zur Programmierung unter Windows und hat eine eingebaute Unterstützung von C++.

2.4 Terrain

Ein Terrain im Sinne dieser Studienarbeit ist ein gleichmäßiges Gitter aus Punkten (3D-Koordinaten), deren Höhenwerte mittels mehr oder weniger komplexen Algorithmen verändert werden, um so eine optisch möglichst natürlich aussehende Landschaft nachzubauen. Solch eine Struktur ist äußerst Effizient was den benötigten Speicherplatz angeht, da lediglich die y Koordinate gespeichert werden muss. Die x und z Koordinaten ergeben sich aus dem Index des Punktegitters. So etwas wird in der Grafikprogrammierung in der Regel *Heightmap* genannt. Für diese Studienarbeit wurde mehrere kleine *Heightmap*s anstatt einer großen implementiert, die im Text als Terrainfelder bezeichnet werden. Ein Terrainfeld beeinhaltet ein gewisse Menge an Zeilen aus Punkten. Mittels dieser Punkte werden die Dreiecke erzeugt.

3 Haupteigenschaften

3.1 Zufällig generiertes Terrain

Es gibt viele mehr oder weniger bekannte Ansätze um ein zufallsgeneriertes Terrain zu erstellen. Einer der bekanntesten ist der *Diamond Square Algorithmus*([DeLoura2000], S.503ff). Bei diesem werden den Eckpunkten einer *Heightmap* zufällige Höhenwerte zugewiesen (Abbildung 1). Diese vier Punkte werden im sogenannten (*Diamond-Step*, Abbildung 2) gemittelt

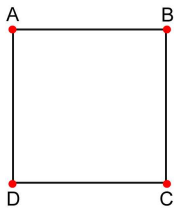

Abbildung 1: Eckpunkte

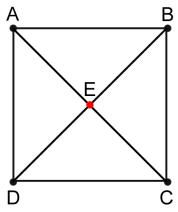

Abbildung 2: Diamond-Step

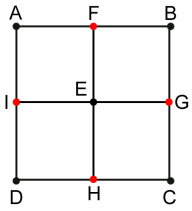

Abbildung 3: Square-Step

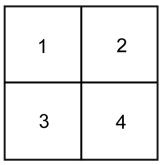

Abbildung 4: Rekursion

und ein Zufallswert aufaddiert, um dem Mittelpunkt der *Heightmap* einen Wert zuzuweisen . Danach werden die Mittelpunkte der Ränder aus den jeweils zwei benachbarten Eckpunkten und dem Mittelpunkt wiederum durch Mittelung und Addition eines Zufallswertes errechnet (*Square-Step*, Abbildung 3). Nun kann die *Heightmap* anhand der errechneten Punkte in vier kleinere *Heightmaps* unterteilt werden, für die der Vorgang wiederholt wird, bis die gewünschte Rekursionstiefe erreicht wurde (Abbildung 4). Je tiefer die Rekursion, desto detailreicher wird das Terrain.

Ein anderer interessanter Ansatz ist *Fault Formation* ([DeLoura2000], S.499ff), welcher Naturkräfte wie die Trennung tektonischer Platten und Erosion simulieren soll. Dazu wird zufällig eine Linie durch eine leere *Heightmap* gezogen. Dann werden alle Werte auf einer Seite der Linie um einen bestimmten Wert erhöht. Es müssen ziemlich viele dieser Linien erzeugt werden, um ein adäquates Ergebnis zu erreichen. Ein anschließendes Weichzeichnen rundet die Kanten im Terrain im Sinne natürlicher Erosion ab.
Obwohl die Ergebnisse dieses Ansatzes optisch äußerst ansprechend sind, kam er für diese Studienarbeit nicht in Frage, weil er leider auch relativ langsam und daher nicht für Echtzeit-Anwendungen geeignet ist.

Es gibt noch viele weitere Ansätze, die fast alle gemeinsam haben, dass ein quadratisches Punktegitter benutzt werden muss, dessen Seitenlängen Zweierpotenzen sind. Dies hat zur Folge, dass das Terrain in Form von mehreren Feldern erstellt werden muss. Eine Unterteilung in Felder bringt diverse Vorteile mit sich, z.B. kann mittels der Felder direkt für ganze Bereiche des Terrains getestet werden, ob diese überhaupt im Sichtbereich der

Abbildung 5: Eine Linie

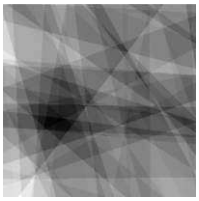

Abbildung 6: Viele Linien

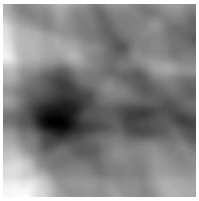

Abbildung 7: Weichgezeichnet

Kamera liegen und vielleicht gar nicht erst gezeichnet werden müssen. Allerdings gibt es auch Nachteile, z.B. muss ein Übergang zwischen den einzelnen Feldern geschaffen werden, damit keine unschönen Sprünge im Terrain zu sehen sind. Desweiteren haben Felder den hässlichen Effekt, dass die Kamera scheinbar auf ein Ende des Terrains zufliegt, doch dann erscheint plötzlich aus dem nichts ein neuer Geländeabschitt. Dies ließe sich jedoch durch die Simulation von Nebel oder dem Einsatz eines falschen Horizontes umgehen.
Für diese Studienarbeit wurde allerdings ein eigener Ansatz erdacht, der die Dreiecke des Terrains zeilenweise statt felderweise aufbaut. Damit ist ein fließender Übergang garantiert (da jede Zeile auf der vorigen Zeile aufbaut und sich dabei nahtlos an diese anschließt) und die extremen Sprünge beim Auftauchen der Felder sind auch beseitigt (weil einzelne Zeilen von Dreiecke so dünn sind, dass deren Auftauche nicht als Sprung wahrgenomen wird). Im Laufe der Implementierung wurde jedoch wieder auf eine Felderstruktur zurückgegriffen, um deren Vorteil der einfacheren Behandlung ganzer Terrainabschnitte nutzen zu können. Innerhalb der Felder wird das Terrain jedoch weiterhin zeilenweise aufgebaut, sodass zumindest der relativ fließende Übergang erhalten bleibt.
Der zeilenweise Aufbau benötigt lediglich Kenntnisse über die zu ändernde Zeile und deren beiden vorangehenden Zeilen. Für jeden Punkt einer Zeile wird geprüft, ob das Terrain davor gestiegen oder gefallen ist. Dazu wird die Differenz der beiden Punkte mit der gleichen X-Koordinate aus den beiden vorigen Terrainzeilen genommen. Bei einem Anstieg wird die Wahrscheinlichkeit auf einen erneuten Anstieg höher gesetzt als auf einen Abstieg und umgekehrt. Dann wird entsprechend ein Zufallswert aufaddiert oder subtrahiert. Für diesen Aufbau sind fünf Variablen im Terraincode

von zentraler Bedeutung: *RandomFactor*, *ChanceDivisor*, *LowerLimit*, *HigherLimit* und *SmoothFactor*. Der *RandomFactor* gibt an, um wieviel ein Punkt im Vergleich zu seinem Vorgänger angehoben bzw. gesenkt werden kann. Dieser Wert wird jedoch durch den *ChanceDivisor* verringert, um die Wahrscheinlichkeit eines An- bzw. Abstieges zu steuern. Hierzu ein Codebeispiel das zeigt, was dem Höhenwert des vorrangehenden Punktes hinzugefügt wird, wenn das Terrain eine höhere Wahrscheinlichkeit für einen Anstieg haben soll:

$$newValue = oldValue - rf/cd + (rand() * rf)/RAND\_MAX);$$

RandomFactor (rf) dividiert durch den *ChanceDivisor* (cd) wird von dem alten Wert subtrahiert und ein zufälliger Wert von Null bis *RandomFactor* wird aufaddiert. Sei *RandomFactor* = 1 und *ChanceDivisor* = 4, dann würde der Punkt erst um 0.25 verringert werden und hinterher um maximal 1.0 (und damit insgesamt um bis zu 0.75) erhöht. Ergebnis: Der neue Punkte liegt mit einer Wahrscheinlichkeit von 25% unter dem alten Punkt und zu 75% darüber.
LowerLimit gibt an, unterhalb welchem Höhenwert die Chance auf einen Anstieg des Terrains höher sein soll als auf einen Abstieg, unabhängig davon, ob das Terrain vorher anstieg oder nicht. *HigherLimit* garantiert dagegen, dass das Terrain oberhalb eines bestimmten Höhenwertes eher wieder abfällt. Diese beiden Werte sorgen dafür, dass das Terrain aufgrund einer Verkettung unglücklicher Zufallswerte nicht ins Bodenlose fällt oder extreme Höhen erreicht.
Anschließend wird jede Zeile noch weichgezeichnet, indem der Höhenwert jedes Punktes einer Zeile mittels einer Gewichtung seinen Nachbarpunkten etwas angepasst wird. Dafür ist der *SmoothFactor* zuständig. Er gibt an, welche Gewichtung der errechnete Wert und seine Nachbarwerte letztlich haben (siehe Codebeispiel in Listing 1). Die Weichzeichnung ist bei diesem Algorithmus besonders wichtig, da sonst nur die Werte innerhalb einer Spalte aufeinander aufbauen, ohne Bezug zu den Nachbarspalten, was in einem extrem stacheligen Terrain enden würde.

Listing 1: Codebeispiel zur Weichzeichnung

```
void TerrainField::smoothRow(int smoothFactor, int row)
{
        //left to right
        for(int x = 1; x < mWidth-1; ++x)
        {
                mVertices[x + row * mWidth]->getCoordinates()->setY(
                        mVertices[x + row * mWidth]->getCoordinates()->getY()
                                * (1 - smoothFactor * 0.01) +
                        mVertices[x - 1 + row * mWidth]->getCoordinates()->getY()
                                * (smoothFactor * 0.01));
        }
```

```
        //right to left
        for(int x = mWidth-2; x > 0; --x)
        {
                mVertices[x + row * mWidth]->getCoordinates()->setY(
                        mVertices[x + row * mWidth]->getCoordinates()->getY()
                                * ( 1 - smoothFactor * 0.01) +
                        mVertices[x + 1 + row * mWidth]->getCoordinates()->getY()
                                * (smoothFactor * 0.01));
        }
}
```

3.2 Dynamisch generiertes Terrain

Das Terrain wird zwar weitgehend zufällig erstellt, dennoch sollte dem Benutzer die Möglichkeit gegeben werden, auf die Terrainentwicklung Einfluss zu nehmen. Um dies in Form eines spielerischen Elements zu bringen, werden Objekte erstellt, die vom Benutzer eingesammelt werden können. Jedes dieser sogenannten *TerraformItem*s bekommt bei seiner Erstellung sinnvolle Werte für die fünf zentralen Variablen *RandomFactor*, *ChanceDivisor*, *LowerLimit*, *HigherLimit* und *SmoothFactor* zugewiesen. Sinnvoll bedeutet in diesem Fall, dass das mit diesen Werten erstellte Terrain nicht ein völliges Chaos aus Ecken und Kanten wird. Um dies zu garantieren, werden die Werte nicht zufällig erstellt, sondern mehrere Wertekombinationen wurden vorgegeben, aus denen jeweils eine ausgewählt wird. Wenn der Benutzer ein *TerraformItem* durchfliegt und somit einsammelt, werden seine Werte ausgelesen und als neuer Standard für die weitere Terrainerstellung genutzt. Abbildung 8 zeigt einen Ausschnitt eines Screenshots mehrerer *TerraformItem*.

Desweiteren hat der Benutzer die Möglichkeit, die Terrainentwicklung direkt über die Regler der GUI zu steuern. Jedoch müssen die Möglichkeiten etwas ausgetestet werden, da viele Einstellungen zu hässlichen Ergebnissen führen können (siehe Abbildung 9).

3.3 Endlos generiertes Terrain

Da Speicher auf jedem System begrenzt ist, kann nicht einfach für neue Terrainfelder neuer Speicherbereich alloziiert werden. Auch wenn zeitgleich der Speicher eines alten Terrainfeldes wieder freigegeben würde, wäre dies keine gute Lösung, da die ständig neue Reservierung und Freigabe von Speicher unnötig Rechenzeit in Anspruch nehmen würde . Stattdessen werden beim Programmstart eine feste Anzahl an Terrainfeldern erstellt. Diese werden in Rotation mit neuen Werten belegt. Das geschieht immer dann, wenn der Benutzer ein Terrainfeld komplett überflogen hat und dieses somit nicht mehr im Sichtbereich ist. Um für einen nahtlosen Übergang zwischen den Terrainfeldern zu sorgen, wird die letzte Reihe an Dreiecken in

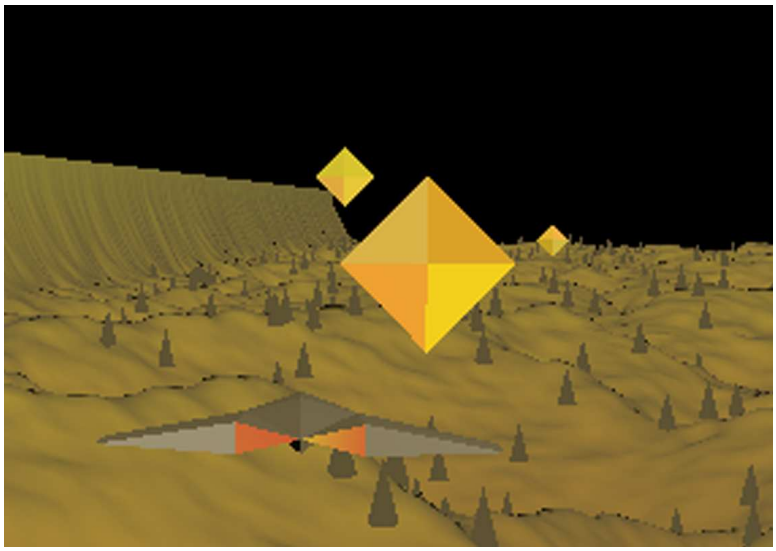

Abbildung 8: TerraformItems

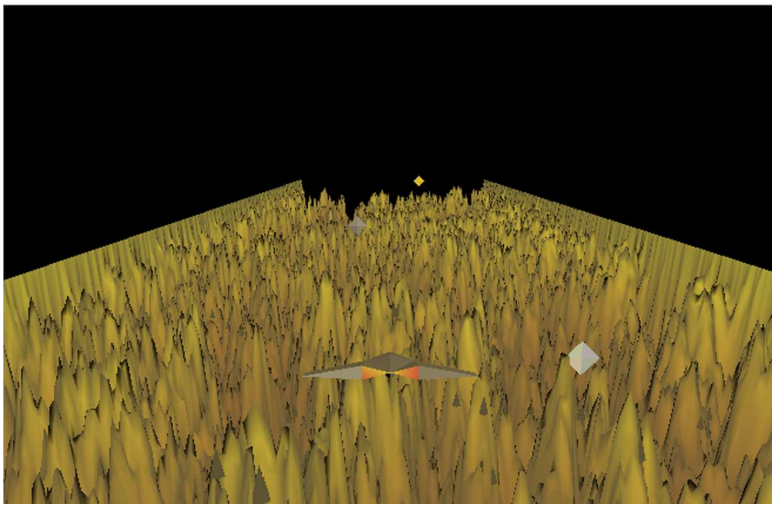

Abbildung 9: Beispiel für schlecht gewählte Terrainwerte

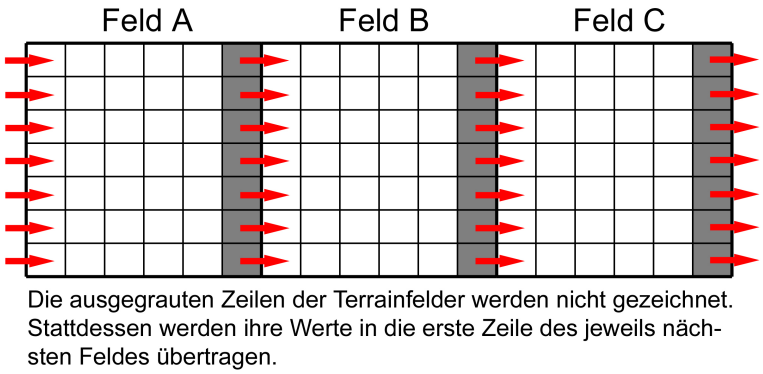

Abbildung 10: Grafik zum Übergang zwischen den Terrainfeldern

jedem Feldes nicht gezeichnet. Stattdessen werden die Höhenwerte dieser Dreiecke auf das nachfolgende Terrain übertragen. Auf diesen aufbauend, wird dann wieder der zeilenweise Zufallsalgorithmus benutzt, um die restlichen Zeilen des neuen Terrainfeldes mit Werten zu belegen. Listing 2 zeigt den Code der Methode *updateField()*, die genau dafür zuständig ist. Abbildung 10 stellt den Übergang zwischen den Terrainfeldern vereinfacht dar. Theoretisch überlappen sich die Terrainfelder also ein wenig, aber da die letzte Reihe jedes Feldes nicht gezeichnet wird, entsteht kein Mehraufwand in Form von Dreiecken die zweimal gezeichnet werden.
Es kommt hierbei vielleicht die Frage auf, warum nicht einfach stattdessen die erste Zeile jedes Feldes auf der letzten Zeile des vorigen Feldes aufgebaut wird. Die Antwort darauf ist, dass dann quasi eine Reihe von Dreiecken 'zwischen' den Feldern gezeichnet werden müsste. Diese wären in der Datenstruktur der Felder schwerlich unterzubringen, da eine Schleife, die über alle Punkte eines Feldes ginge, ihren Start nicht beim Index 0 hätte sondern theoretisch in der letzten Zeile des vorigen Feldes. Dies wäre äußerst unpraktisch bei der Implementierung und brächte außerdem das Problem mit sich, dass die Punktereihe des vorigen Feldes bei dessen Neuberechnung ja schon mit neuen Werten überschrieben würde, während die 'Zwischendreiecke' noch auf die alten Werte angewiesen wären.

Listing 2: Codebeispiel zu *updateField()*

```
void Terrain::updateField(int field)
{
        int oldField;
        if(field > 0)
        {
                oldField = field-1;
        }else
        {
                oldField = mFieldCount-1;
        }
```

```
        int v = 0;

        for(int i = (mTerrainLength-2)*mTerrainWidth; i < mTerrainLength*mTerrainWidth; ++i)
        {
                *(mFields[field]->getVertice(v)->getCoordinates()) =
                        *(mFields[oldField]->getVertice(i)->getCoordinates());
                ++v;
        }
        mFields[field]->update(mRandomFactor, mSmoothFactor, mChanceDivisor,
                mLowerLimit, mHigherLimit);
}
```

4 Weitere Features

4.1 Steuerung

Der Benutzer sollte die Möglichkeit haben, ein kleines Fluggerät über das Terrain steuern zu können. Auf ein realistisches und physikalisch korrektes Flugverhalten wurde verzichtet, da der Fokus dieser Arbeit auf dem Terrain lag und nicht auf der Erschaffung eines Flugsimulators. Lediglich das genretypische Invertieren der vertikalen Steuerung wurde beachtet. Wird die Maus vom Benutzer zu sich herangezogen (er bewegt diese quasi nach unten), so ändert das Fluggerät seine Flugrichtung nach oben statt nach unten, änlich einem Steuerknüppel eines Flugzeuges.

Bei einer ersten Implementierung einer Steuerung über die Tastatur stellte sich Tastaturabfrage von *Qt* schnell als zu langsam und unpraktisch heraus. Deshalb wurde eine Steuerung über die Maus implementiert. Damit der *Center of Interest* und somit die Ausrichtung des Fluggerätes einer gleichmäßigen Kreisbahn folgen, wurden Kugelkoordinaten (Abbildung 11) für die Steuerung genutzt (eine Erweiterung von Polarkoordinaten). Diese nutzen zwei Winkel und einen Radius, um die Koordinaten eines beliebigen Punktes auf einer kugelförmigen Fläche um den Ausgangspunkt zu bestimmen. Der Winkel Phi gibt dabei die horizontale Abweichung an, während der Winkel Theta die vertikale Abweichung angibt. Seien x, y und z die Koordinaten des *Center of Interest* und der Ausgangspunkt liege im Ursprung bei (0,0,0), dann berechnen sich die Kugelkoordinaten wie folgt:

$$r = \sqrt{x^2 + y^2 + z^2};$$

$$\varphi = \arccos \frac{x}{\sqrt{x^2 + y^2}} \text{ für } y \geq 0;$$

$$\varphi = 2\pi - \arccos \frac{x}{\sqrt{x^2 + y^2}} \text{ für } y < 0;$$

$$\theta = \frac{\pi}{2} - \arctan \frac{z}{x^2 + y^2};$$

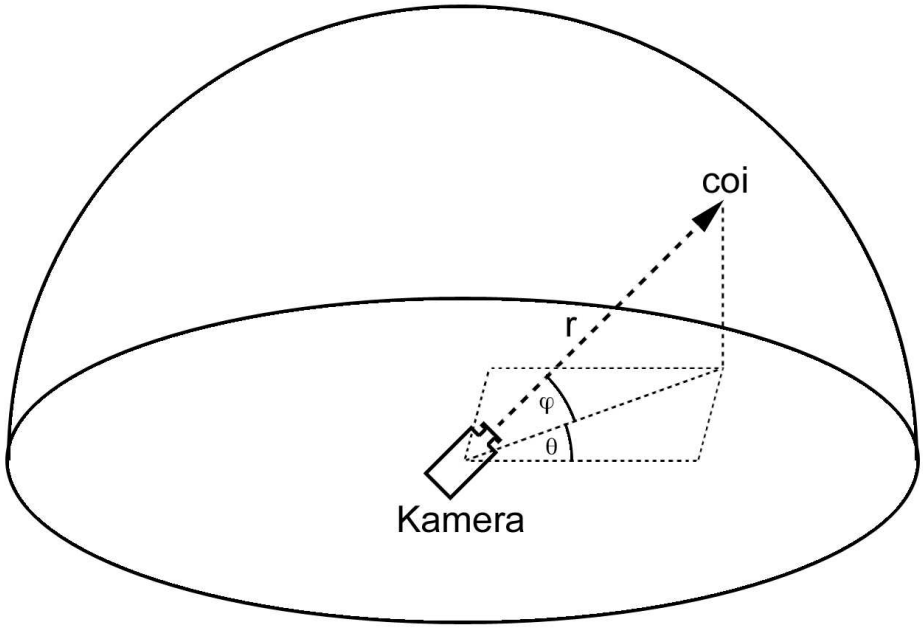

Abbildung 11: Polar- bzw. Kugelkoordinaten

Nun können Phi und Theta je nach Mausbewegung oder Tastatureingabe vergrößert oder verkleinert werden. Anschließend müssen die Kugelkoordinaten wieder in kartesische Koordinaten umgerechnet werden:

$$x = r \sin\theta \cos\varphi;$$

$$y = r \sin\theta \sin\varphi;$$

$$z = r \cos\theta;$$

Die Variablen x, y und z beschreiben nun den *Center of Interest* nach der Richtungsänderung. Da es dem Benutzer nicht gestattet sein sollte umzudrehen und zurückzufliegen, wurde die Abweichung, die die beiden Winkel von der Ursprungsausrichtung (entlang der negativen Z-Achse) annehmen können, eingeschränkt.
Solange die eingestellte Fluggeschwindigkeit größer als Null ist und keine Kollision mit dem Terrain erkannt wurde, bewegen sich die Kamera und das Fluggerät in Blickrichtung vorwärts.

4.2 Kollisionserkennung

Ein wichtiger Punkt für die Grundelemente des Programms ist die Kollisionserkennung. Ohne Kollisionserkennung könnte der Benutzer einfach

das Terrain rammen und durchfliegen ohne daran hängenzubleiben. Außerdem könnte er keine Objekte einsammeln um Einfluss auf die Terrainbildung zu nehmen. Um zu ermitteln ob die Flugbahn des Fluggerätes das Terrain oder ein Objekt kreuzt, müssen Schnitttests zwischen dem Richtungsvektor, der die Blickrichtung beschreibt, und den Dreiecken des Terrains bzw. der Objekte durchgeführt werden. Dazu wurde auf den relativ bekannten Schnitttest von Thomas Moeller und Ben Trumbore ([MöllerTrumbore1997]) zurückgegriffen.
Das Hauptproblem bei Schnitttesst ist jedoch meistens nicht der Schnitttest selbst, sondern die extrem große Anzahl an Schnitttests die durchgeführt werden müssen, im Prinzip nämlich soviele wie es Dreiecke in der Szene gibt, da all diese ja theoretisch getroffen werden können. Dieser *brute-force* Ansatz ist entschieden zu zeitaufwendig und somit für eine Echtzeitanwendung ungeeignet. Eine weit verbreitete Lösung dieses Problems ist die Unterteilung der Szene in kleinere Abschnitte, um dann nur die Dreiecke in den Abschnitten testen zu müssen, durch die der Richtungsvektor verläuft. Dazu werden meist Baumstrukturen verwendet, wie z.B. ein sogenannter *KD-tree*. Dabei werden die Dreiecke oder Punkte reihum nach k Dimensionen unterteilt (*KD-tree* steht für *k-dimensional tree*. Um dann eine Kollisionserkennung mit der Szene durchzuführen, müssen nur noch wenige Schnitttests für die zu durchlaufenden Knoten des Baumes gemacht werden bis man in den Blättern landet. Dort verbleiben dann relativ wenige Schnitttest mit den Dreiecken die den erreichten Blättern zugeordnet sind. Je größer die zugrunde liegende Menge an Dreiecken ist, desto mehr Zeit spart man mit einer solchen Baumstruktur ein. Das Problem ist jedoch, dass sich die Szene in diesem Programm ständig verändert. Deshalb müsste die Baumstruktur ständig neu aufgebaut werden, was einen nicht unerheblichen Zeitaufwand mit sich brächte.
Um echtzeitfähig zu bleiben wurde stattdessen ein eigener Ansatz entwickelt. Dieser trifft eine Vorhersage bezüglich der Zielkoordinaten des Fluggerätes für den Fall, dass keine Kollision auftritt und prüft hinterher, ob dies zutrifft. Als erstes werden mittels der Startposition, der Blickrichtung und der momentanen Fluggeschwindigkeit die Zielkoordinaten errechnet. Dann wird die Startposition und die Zielkoordinaten verglichen, um minimale und maximale x und z Koordinaten für ein Rechteck zu erhalten, welches die errechnete Flugbahn komplett einschließt. Dazu werden die den minimalen und maximalen Koordinaten am nähesten liegenden Punkte gesucht. Dabei wird ausgenutzt, dass die x und z Koordinaten der Punkte nicht willkürlich im Raum liegen, sondern auf den ganzen Zahlen des Koordinatensystems. Von daher reicht es aus, einfach die Koordinaten auf ganze Werte auf- oder abzurunden. So wird ein Gebiet eingegrenzt, dass alle Dreiecke einschließt, die potentielle Kandidaten für eine Kollision sind (siehe Abbildung 12 und dazu Listing 3).

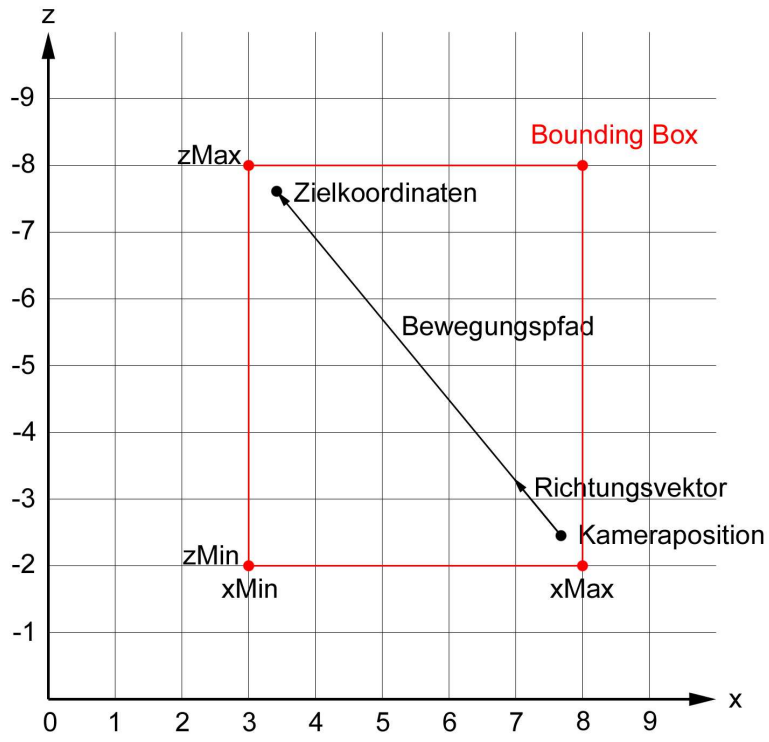

Abbildung 12: Bounding Box um den Bewegungspfad

Listing 3: Codebeispiel zur Bounding Box

```
//calculate the highest and lowest x value
if(mCameraPosition->getX() < mDestination->getX())
{
        xMin = floor(mCameraPosition->getX());
        xMax = ceil(mDestination->getX());
}else
{
        xMin = floor(mDestination->getX());
        xMax = ceil(mCameraPosition->getX());
}

//cut x if it gets out of the borders
if(xMin < 0)
{
        xMin = 0;
}
if(xMax > mTerrainWidth-1)
{
        xMax = mTerrainWidth-1;
}

//calculate the highest and lowest z value
zMin = floor(mCameraPosition->getZ() * (-1));
zMax = ceil(mDestination->getZ() * (-1));

//modulo to adjust to the number of rows in a field
zMin = (zMin % (mTerrainLength-1));
zMax = (zMax % (mTerrainLength-1));
```

Die Werte können nun als Start- und Endwerte für eine Schleife genutzt werden, um die entsprechenden Dreiecke zu ermitteln. Dies wird durch die Datenstruktur möglich, in der jeder Punkt weiß, zu welchen Dreiecken er gehört. Mit den so erhaltenen Dreiecken können dann die Schnitttests durchgeführt werden.
Wurde keine Kollision gefunden, können die Kamera und das Fluggerät einfach vorwärts geschoben werden. Wurden eine oder mehrere Kollisionen gefunden, wird die Bewegung eingestellt. Der Benutzer kann nun die Flugrichtung ändern und die Geschwindigkeit wieder erhöhen um den Flug fortzusetzen.

4.3 GUI

Zu Beginn der Studienarbeit wurde *GLUT* (*OpenGL Utility Toolkit*) benutzt, um die mit *OpenGL*-Befehlen erstellte 3D-Szene in einem Fenster darzustellen. *GLUT* ist ähnlich wie das anfänglich vorgestellte *Qt* eine Schnittstelle zur Programmierung mit *OpenGL*. Da *GLUT* selbst aber keine GUI zur Verfügung stellt und die Programmierung einer eigenen GUI zu aufwendig ist, sollte hierbei auf *Qt* zurückgegriffen werden. Da mit der Implementation der GUI jedoch erst begonnen wurde, als das *OpenGL* Grundgerüst mit *GLUT* schon stand, traten einige unerwartete Probleme auf. *Qt*

bietet nämlich nicht einfach nur zusätzliche Befehle an, sondern ist wie *GLUT* selbst als Grundgerüst ausgelegt. Deswegen musste zu diesem Zeitpunkt die Grundstruktur des Programms abgeändert werden, was wertvolle Zeit kostete. Desweiteren verbietet *Qt* den Gebrauch diverser Basismethoden von *GLUT* und ersetzt diese durch eigene Varianten. Diese sind sich zwar sehr ähnlich, doch fand sich u.A. kein Pendant zur bei *GLUT* sehr nützlichen *IdleFunc*. Diese wird üblicherweise benutzt, um die Zeichenmethode beständig aufrufen zu lassen, was z.B. eine kontinuierliche Bewegung durch das Terrain ermöglichte. Hier fand sich in *Qt* dann nach einiger Suche auf diversen Webseiten der Hinweis auf eine Timer-Funktion, mit das Gleiche erreicht werden konnte. Es wurde ein Timer erstellt, der alle 40 Millisekunden ein Signal auslöst. Dies wurde genutzt, um im Prinzip den gleichen Effekt wie bei der *IdleFunc* zu erreichen. Der Intervall von 40 Millisekunden wurde gewählt, um eine Framerate von 25 Bildern pro Sekunde zu erzeugen. Diese variiert aber natürlich mit der Leistungsfähigkeit des Systems, auf der das Programm läuft.
Die GUI sollte dem Benutzer diverse Informationen über den Zustand des Terrains liefern. Umgesetzt wurde dies in Form von mehreren Displays, die die momentanen Werte der zentralen Variablen des Programms anzeigen. Zur Veränderung dieser Variablen bei laufendem Programm wurden den Displays Schieberegler zugefügt. Dies erleichterte die Entwicklung des Terrain-Algorithmus enorm, da es nicht mehr nötig war, das Programm wegen jeder kleinen Änderung an diesen Variablen neu zu kompilieren. Da es für den Benutzer wohl auch interessant sein würde die Terrainentwicklung direkt per Schieberegler zu kontrollieren als nur über das spielerische Element der einzusammelnden Objekte, wurden die Schieberegler ein fester Teil der GUI. Die Grenzwerte der Regler wurden so gewählt, dass viele unsinnige Werte nicht eingestellt werden können. Dennoch sind einige Tests erforderlich, um auf optisch ansprechende Wertekombinationen zu kommen. Einen Screenshot der GUI zeigt die Abbildung 13.

4.4 Farbgebung

Die Festlegung der Farbwerte erwies sich durch das Zufallselement des Terrains als schwierig. Da es keine vorgefertigten Geländeteile gibt, war eine manuelle, detaillierte Farbgebung nicht möglich. Stattdessen blieb nur der Weg, die Farbgebund den Höhenwerten anzupassen. Dabei wurde auch versucht, dem Farbwert ein Zufallselement zu geben, dies resultierte allerdings in unschönen Farbsprenkeln anstatt weicheren Übergängen und wurde deshalb wieder entfernt. Die Farbe wird nun für jeden Punkt einzeln, wie im Codebeispiel in Listing 4, ermittelt. Die Division durch 100 kommt daher, dass sich die Höhenwerte des Terrains bei den Standardeinstellungen der Werte für *LowerLimit* und *HigherLimit* zwischen 0 und 100 bewegen, die Funktion *glColor3f* allerdings Werte zwischen 0 und 1 er-

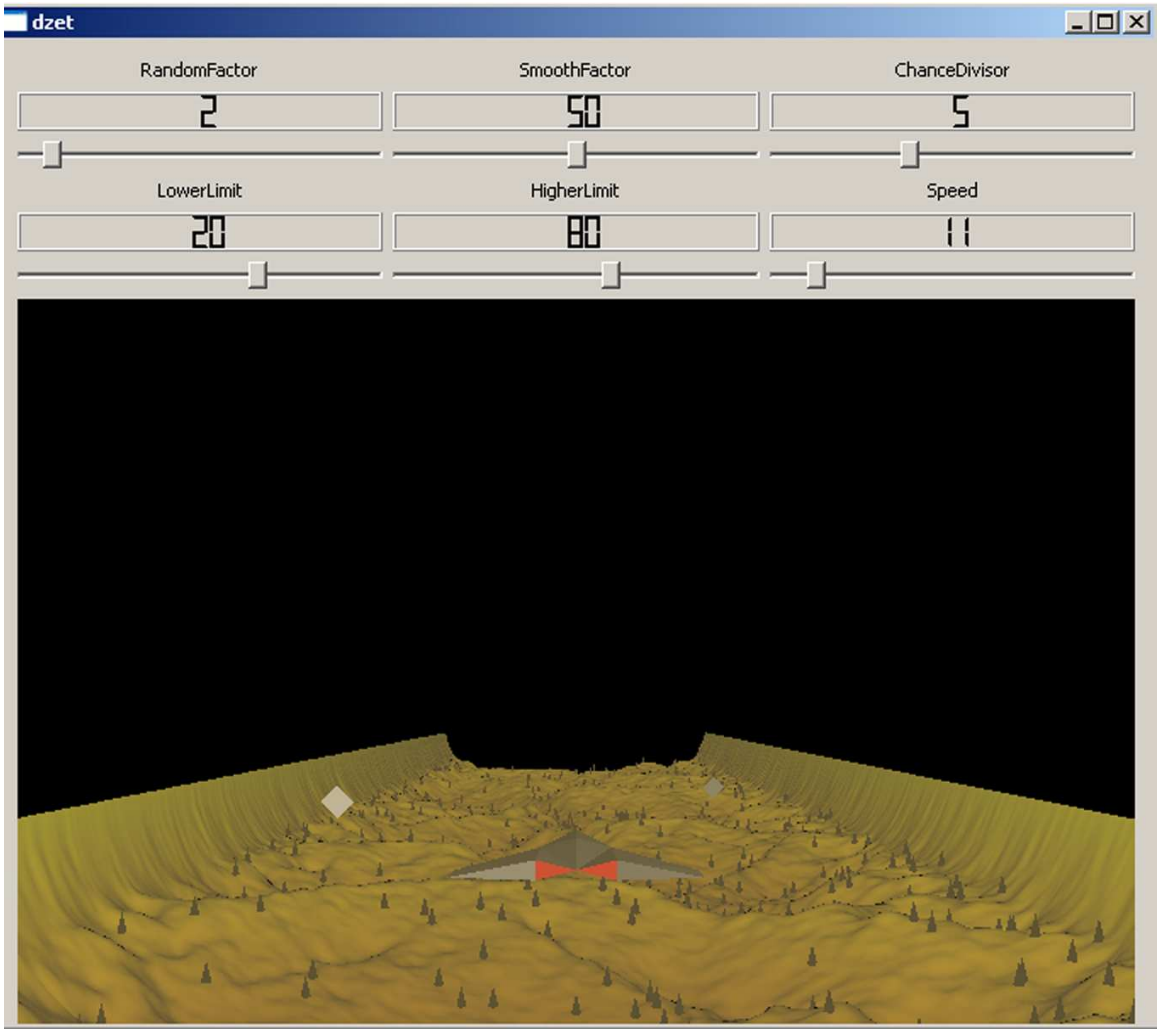

Abbildung 13: Die Bedienelemente

wartet. Damit Farben auch richtig dargestellt werden, ist eine Beleuchtung nötig. Dazu wurde eine direktionale Lichtquelle erstellt, die das Terrain stets von schräg oben beleuchtet. Die genauen Werte der verschiedenen Parameter der Lichtquelle und der für die Beleuchtung zu definierenden Materialeigenschaften können im Codeausschnitt im Listing 5 eingesehen werden.

Listing 4: Codebeispiel zur Farbgebung

```
void Triangle::setHeightColor(float y)
{
        float color = y/100;
        glColor3f(color, color * color, 0.0);
}
```

Listing 5: Codebeispiel zur Beleuchtung

```
glEnable(GL_LIGHTING);
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,0);
        glEnable(GL_COLOR_MATERIAL);
        glColorMaterial(GL_FRONT,GL_DIFFUSE);

        //settings for the single lightsource
        glEnable(GL_LIGHT0);
        float lightAmbient[] = { 0.0f, 0.0f, 0.0f, 1.0f };
        float lightDiffuse[] = { 0.5f, 0.5f, 0.5f, 1.0f };
        float lightSpecular[] = { 0.5f, 0.5f, 0.5f, 1.0f };
        float lightPosition[] = { 1.0, 0.25, -0.5, 1.0 };
        float lightDirection[] = { -1.0, -0.25, 0.5 };
        glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, lightSpecular);
        glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
        glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, lightDirection);

        //settings for the material
        float matAmbient[] = { 0.25f, 0.20f, 0.07f, 1.0f };
        float matDiffuse[] = { 0.75f, 0.61f, 0.23f, 1.0f };
        float matSpecular[] = { 0.63f, 0.56f, 0.37f, 1.0f };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, matAmbient);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, matDiffuse);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, matSpecular);
}
```

5 Sonstiges

5.1 Generelle Programmstruktur

Im Folgenden soll erklärt werden, was bei Programmstart passiert und was die einzelnen Klassen machen (siehe dazu das unvollstä Klassendiagramm in Abbildung 14. Unvollständig weil die Klassen *Vector3D* und *Triangle* weggelassen wurden, da fast alle Klassen diese irgendwie nutzen und das Diagramm damit zu unübersichtlich geworden wäre) : Initial wird wie

bei jedem Programm die Main-Methode aufgerufen, diese lässt *Qt* dann erstmal das Objekt *MainWidget* vom Typ *QWidget* erstellen, sozusagen das Hauptausgabefenster. Dieses erstellt ein weiteres *QWidget*-Objekt namens *RenderingWidget* welches auch Eigenschaften von *QGLWidget* erbt und damit für die Hauptfunktionen von *OpenGL* zuständig ist. Desweiteren erstellt *MainWidget* sämtliche GUI-Elemente.
RenderingWidget erstellt ein Objekt vom Typ *Terrain* sowie eines vom Typ *Ship*. Außerdem legt es den *OpenGL* Kontext fest, d.h. es setzt die Kamera, die Größe des Ausgabefensters, die Lichtquelle(n) und die generellen Materialeigenschaften. Seine Methode *paintGL()* bildet das Herzstück des Programms. Diese wird von einem Timer ständig neu aufgerufen um die Position der Kamera und des Fluggerätes zu aktualisieren. Sie leitet falls nötig ein Update der Terrainfelder ein und sorgt dafür, dass alle Dreiecke gezeichnet werden. Die Annahme von Maus- und Tastaturbefehlen wird ebenfalls vom *RenderingWidget* übernommen.
Ship initialisiert die Dreiecke, welche das Fluggerät bilden.
Die Klasse *Terrain* verwaltet die zentralen Variablen des Terrainalgorithmus (*RandomFactor*, *ChanceDivisor*, *LowerLimit*, *HigherLimit* und *SmoothFactor*) und erzeugt die Terrainfelder in Form von *TerrainField*-Objekten. Sobald *RenderingWidget* das Signal gibt, dass die Felder rotiert werden müssen, prüft *Terrain* welches das älteste Feld ist und überschreibt dessen ersten zwei Terrainzeilen mit den letzten beiden Zeilen aus dem jüngsten Feld. Dann leitet es die Neuberechnung der restlichen Zeilen des alten Feldes ein und übergibt dabei die aktuellen Werte der zentralen Variablen.
Die *TerrainField*-Objekte wissen theoretisch nichts voneinander, da sie von *Terrain* verwaltet werden. Jedes *TerrainField* kümmert sich um seine Datenstruktur aus *Vertice*s an sich, sowieso die dem Feld zugehörigen Bäume (*TreeItem*s) und *TerraformItem*s. Wichtig ist vor allem, dass *TerrainField* die *Vertice*s und Dreiecke (Klasse *Triangle*s) so miteinander verlinkt, dass jedes *Vertice* weiß, zu wievielen und welchen Dreiecken es gehört.
Die Klasse *Vertice* speichert für jeden Punkt dessen Koordinaten, die Dreiecke zu denen er gehört und die Normale in diesem Punkt, die sich aus der gemittelten Normalen aller seiner Dreiecke ergibt. Dies ist für eine weiche Beleuchtung per Vertex nötig anstatt einer kantigen Beleuchtung per Dreieck.
Triangle speichert alle Daten bezüglich der Dreiecke. Das wären deren Koordinaten, die Flächennormale und die Normalen der Punkte. Desweiteren besitzt die Klasse die Methode zur Durchführung von Schnitttests sowie Methoden zur Farbwahl und Zeichnung des Dreiecks.
*TerraformItem*s speichern Daten über ihre Koordinaten, eine Wertekombination für die zentralen Variablen und ihre Farbe (damit es möglich ist zu erkennen, welche Wertekombination ein *TerraformItem* momentan gespeichert hat). Die virtuelle Klasse *Item* bietet einen gemeinsam Rumpf für diverse Subklassen wie *TreeItem*, damit diese von anderen Klassen auf die

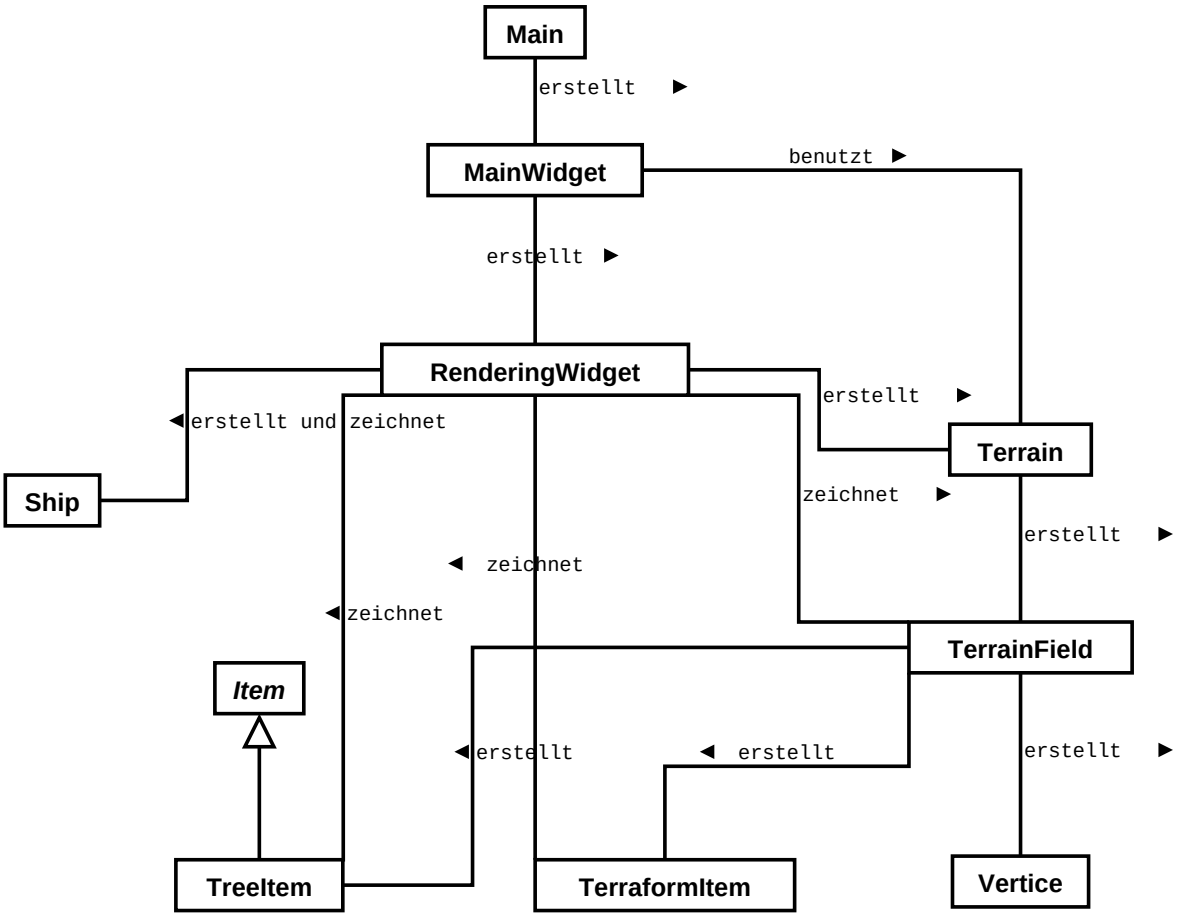

Abbildung 14: Ein unvollständiges Klassendiagramm

gleiche Weise angesprochen werden können. Damit wurde jedoch nur ein kleiner Grundstein gelegt, da bisher nur *TreeItem* als von *Item* von implementiert wurde und sie dadurch momentan noch ziemlich unnötig ist.
TreeItem ist eine von der virtuellen Klasse *Item* abgeleitete Klasse und definiert deren Methoden zum Zwecke der Erstellung eines simplen Baumes aus wenigen Dreiecken.
Zu guter Letzt gibt es noch die Klasse *Vector3D*, die in fast keinem *OpenGL*-Programm fehlen darf. Diese beschreibt 3-dimensionale Vektoren und stellt die benötigten Methoden zur Verfügung, damit Vektorrechnungen durchgeführt werden können.

5.2 Zufallszahlen generieren

Für die Generierung eines Terrains mit Zufallswerten wird natürlich eine Möglichkeit gebraucht, solche zu erstellen. Das Problem dabei ist, dass es in der Programmierwelt keine wirklichen Zufälle gibt. Es wird daher in der Regel versucht, zumindest den Anschein von Zufallswerten zu erwecken. Als Grundlage dient hierbei die Funktion *rand()* aus der C++ Bibliothek. Diese erzeugt sprunghafte Zahlen zwischen 0 und einem maximalen Wert, der je nach Entwicklungsumgebung variieren kann. Da es jedoch keinen richtigen Zufall gibt, unterliegen die Zahlensprünge eben einem mathematischen Algorithmus. Deswegen erzeugt *rand()* bei jedem Programmstart die gleichen Zahlen in der gleichen Reihenfolge. Damit sähe das Terrain jedesmal exakt gleich aus (zumindest wenn der Benutzer nicht die zentralen Variablen des Terrains verändert). Um dem entgegenzuwirken gibt es die Funktion *srand()*, mit der man *rand()* mit einer beliebigen Zahl initialisieren kann. Für diese Studienarbeit wurde zur Initialisierung die in C++ vorhandene Funktion *time* genutzt, welche die vergangenen Sekunden seit dem 1.Januar, 1970, 0.00 Uhr UTC misst. Mit diesem oft angewandten Vorgehen ist garantiert, dass das Terrain mit jedem Programmstart unterschiedlich aussieht (sofern nicht jemand das Programm während einer Sekunde mehrfach startet).

5.3 SLOTs und SIGNALs

Um die GUI-Elemente mit den jeweiligen Variablen zu verbinden, bietet *Qt SLOT*s und *SIGNAL*s an. Diese bieten eine Alternative zu dem bisherigen Konzept von *Callbacks*. Man definiert z.B. für Bedienelemente der GUI und den Variablen, die von diesen gesteuert werden sollen, jeweils einen oder mehrere *SLOT*(s) und ein *SIGNAL*(s) und verbindet diese. Eine Änderung einer Variable sendet dann über das entsprechende *SIGNAL* dem zugewiesenen *SLOT* des Bedienelements, dass es verändert wurde und das Bedienelement passt seine Anzeige dann an. Ein Beispiel aus diesem Programm:
Die Variable *mSpeed* in der Klasse *RenderingWidget* gibt an, wie schnell sich die Kamera und das Fluggerät momentan über das Terrain bewegen. Für *mSpeed* wurde ein *SLOT* definiert, der die Variable auf einen übergebenen Wert einstellt und ein *SIGNAL* welches Bescheid gibt, falls sich Variable geändert wurde. In der für die GUI zuständigen Klasse *MainWidget* wurde für den entsprechenden Schieberegler *mSliderSpeed* ebenfalls ein *SLOT* und ein *SIGNAL* mit den gleichen Auswirkungen erstellt und mit denen von *mSpeed* verbunden. Sobald nun die Variable *mSpeed* verändert wird, erfährt *mSliderSpeed* dies und nimmt den selben Wert an. Wird der Schieberegler *mSliderSpeed* vom Benutzer manuell verstellt, so erfährt dies *mSpeed* und nimmt ebenfalls den eingestellten Wert an. Listing 6 zeigt beispielhaft

den Codeausschnitt, der die *SLOT*s und *SIGNAL*s der Variable *mSpeed* mit denen ihrer zugehörigen GUI-Elemente *mSliderSpeed* und *mLcdSpeed* verbindet. Listing 7 zeigt die Definition des entsprechenden *SLOT*s für *mSpeed* während Listing 8 die Deklaration des zugehörigen *SIGNAL*s zeigt. In der Tat muss ein *SIGNAL* nicht definiert werden, da seine Auswirkungen quasi durch die ihm zugewiesenen *SLOT*s definiert werden.

Listing 6: Verbinden von SLOTs und SIGNALs

```
//connect the speed slider with the speed display
connect(mSliderSpeed, SIGNAL(valueChanged(int)),
        mLcdSpeed, SLOT(display(int)));
//connect the speed slider with the speed variable
connect(mSliderSpeed, SIGNAL(valueChanged(int)),
        mRenderingWidget, SLOT(setSlotSpeed(int)));
//connect the speed variable with the speed slider
connect(mRenderingWidget, SIGNAL(speedChanged(int)),
        mSliderSpeed, SLOT(setValue(int)));
```

Listing 7: SLOT für *mSpeed*

```
void RenderingWidget::setSlotSpeed(int x)
{
        this->mSpeed = x;
}
```

Listing 8: SIGNAL für *mSpeed*

```
signals:
        void speedChanged(int x);
```

6 Fazit

Das Ergebnis dieser Arbeit erfüllt grundsätzlich die Zielsetzungen, die bei der Einreichung festgelegt wurden. Mittels Zufallsfaktoren wird ein Terrain erzeugt, welches der Benutzer mit einem Fluggerät überfliegen und durch Einsammeln von Objekten verändern kann. Der Schnitttest beeinhaltet eine recht effiziente Methode zur Auswahl der zu testenden Dreiecke und spart sich damit eine sonst anfallende Implementierung von bekannten Baumstrukturen zur Einteilung der Szene. Ein halbwegs fließender Übergang zwischen den verschiedenen Terrainarten ist vorhanden aber sollte noch ausgebaut werden. Zwar passen die Terrainfelder nahtlos aneinander (also keine Sprünge, Kanten oder ähnliches), dennoch geht die Terrainart ziemlich abrupt von beispielsweise relativen flachen Hügeln in steile und kantige Berge über. Das Terrain wird wie gefordert endlos erzeugt, sodass es theoretisch möglich sein sollte, endlose Stunden oder Tage über das Terrain fliegen zu können. Ein Feldtest diesbezüglich war aus

zeitlichen Gründen leider nicht durchführbar. Dem Terrain wurden auch Bäume hinzugefügt, die jedoch eher symbolischer Natur sind. Auf eine detailreichere Flora und weitere Hindernisse wie Felsen wurde zugunsten von wichtigeren Programmteilen verzichtet. Eine einfache GUI stellt dem Benutzer diverse Statusinformationen und Steuerelemente zur Verfügung, mit denen er auch das Terrain verändern kann.

7 Ausblick

Obwohl die Ziele dieser Studienarbeit grundsätzlich erfüllt wurden, bleibt natürlich noch sehr viel Raum für Verbesserungen und Erweiterungen. Einige Elemente wurden aus Zeitgründen nur sehr minimalistisch umgesetzt, so wären z.B. eine Vielzahl weiterer Hinderniss wie Felsen oder ähnliches wünschenswert. Optische Verbesserungen des Terrains durch Änderungen am Algorithmus und der Farbgebung sind denkbar und auch ein wirklich weicher Übergang zwischen verschiedenen Terrainarten wäre schön. Ein weiterer Schritt wäre das Hinzufügen von Texturen, wobei dies bei einem zufällig erzeugten Terrain nicht ganz trivial ist. Man müsste bei der Berechnung der Heighmap für ein *Terrainfeld* eine *Texturemap* miterzeugen, doch dies wäre wohl auch nicht mehr als eine von den Höhenwerten abhängige Farbgebung.
Ein guter Ansatz für Erweiterungen wäre ein weitreichenderes Nutzen der Felderstruktur, um z.B. auch eine seitliche Erweiterung des Terrains zu ermöglichen oder *View Frustum Culling* zu implementieren. Elemente wie *Bump Mapping*, *Shader* oder gar Partikeleffekte für herabstürzende Meteoriten denen der Benutzer ausweichen muss, sind alle theoretisch machbar. Ein Schadensmodell für das Fluggerät, sodass dieses nach ein paar Kollisionen abstürzt, wäre auch interessant.

Abbildungsverzeichnis

Literatur

[DeLoura2000] edited by Mark DeLoura; Game Programming Gems; Charles River Media; 2000; ISBN 1-58450-049-2

[DeLoura2001] edited by Mark DeLoura; Game Programming Gems 2; Charles River Media; 2001; ISBN 1-58450-054-9

[MöllerTrumbore1997] Thomas Möller and Ben Trumbore; Fast, Minimum-Storage Ray-Triangle Intersection; 1997; Journal of Graphic Tools; http://jgt.akpeters.com/papers/MollerTrumbore97/

[Angel2002] Edward Angel; OpenGL - A Primer; 2002; Addison-Wesley; ISBN 0-201-74186-5

[HearnBaker2004] Donald Hearn and Pauline Baker; Computer Graphics with OpenGL, Third Edition; 2004; Pearson Prentice Hall; ISBN 0-13-015390-7

[OpenGL] Silicon Graphics; http://www.opengl.org/

[Herold2001] Helmut Herold; Das QT Buch; 2001; Super Press; ISBN 3-934678-76-9

[QT] Trolltech; http://trolltech.com/products/qt

[Shirley2002] Peter Shirley; Fundamentals of Computer Graphics; Addison Wesley; 2002; ISBN 1-56881-124-1

[Stroustrup2000] Bjarne Stroustrup; The C++ Programming Language, Special Edition; 2000; Addison-Wesley, ISBN 978-0201700732

[GoMiSa2000] Michel Goossens, Frank Mittelbach, Alexander Samarin; Der Latex Begleiter; 2000; Addison-Wesley; ISBN 3-8273-1689-8