Johann Wolfgang Goethe Universität
Frankfurt am Main

Diplomarbeit
Fraktale Planetengenerierung

Jörg Homann ⟨joerg.homann@gmx.net⟩
vorgelegt am 01.02.2006

Prüfer: Prof. Dr.- Ing. D. Krömker
Betreuer: Dipl.-Inform. Tobias Breiner

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Diplomarbeit ohne unzulässige Hilfe und nur unter Verwendung der angegebenen Literatur angefertigt wurde.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Frankfurt am Main, den 30.01.2005

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr.-Ing. Detlef Krömker für die Möglichkeit bedanken, ein so interessantes Diplomarbeitsthema bearbeiten zu dürfen sowie bei Tobias Breiner für die Betreuung dieser Arbeit. Desweiteren möchte ich mich bei Sebastian Schäfer für die Hilfestellung bei zahlreichen Fragen zu C++ und OpenGL, Katja Bretfeld für die Unterstützung während des Studiums bedanken.

Besonderer Dank geht an meine Eltern Marion und Detlef Homann, fürs Korrekturlesen und die Hilfe beim Anfertigen verschiedener Abbildungen sowie die intensive Unterstützung während des gesamten Studiums und der Diplomarbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Darstellung und Visualisierung von realen und imaginären Landschaften in der Computergrafik hat in den letzten Jahren stark an Bedeutung gewonnen. Die Anwendungen reichen dabei von Visualisierung (GIS, 3D-Atlas) über Simulation bis zu Kunst und Unterhaltung. Ziel ist es dabei, dem Betrachter eine realistische und glaubwürdige Landschaft zu präsentieren, unabhängig davon, ob diese reale Vorbilder haben (Erde, Mond, Mars, usw.) oder rein fiktiv sind. Virtuelle Landschaften oder sogar ganze Planeten können dabei auf verschiedene Art und Weise modelliert werden. Eine Möglichkeit ist das Erstellen mit einem 3D-Modellierungsprogramm, welches aber bei ganzen Planeten mit hohen Detailgrad enorm aufwendig ist. Eine andere Alternative ist, das Aussehen dem Zufall zu überlassen und die Welt mit Hilfe von Fraktalen zu erschaffen. Hier hat man zwar kaum Möglichkeiten das Ergebnis zu beeinflußen, dennoch lassen sich damit oft sehr gute Ergbenisse erreichen.

Der Begriff der Fraktale ist untrennbar mit Benoît Mandelbrot verknüpft, welcher in seinem Buch *„The Fractal Geometry of Nature“* [1] als erster erkannt hat, dass viele natürliche Gebilde und Strukturen einen hohen Grad an Selbstähnlichkeit aufweisen. Das beschränkt sich dabei aber nicht nur auf Landschaftsformen und Gebirgsstrukturen, sondern findet sich z.B. auch in Flußsystemen, im Blutkreislauf, im Pflanzen- oder Kristallwachstum wieder. Abbildung 1.1 zeigt die Kochsche Schneeflocke, die zwar ein Fraktal ist, aber große Ähnlichkeiten mit einer realen Schneeflocke besitzt, auch wenn echte Schneeflocken eine größere

Vielfalt und Komplexität aufweisen.

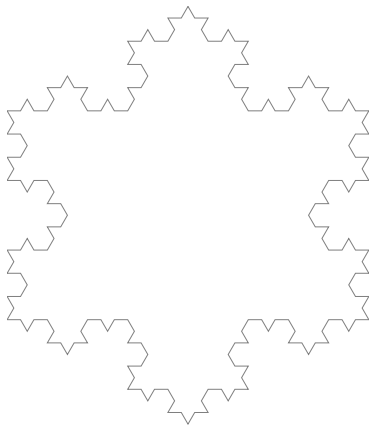

Abbildung 1.1: Kochsche Schneeflocke

Die Ideen Mandelbrots fasste Ken Musgrave, der eine Zeit mit ihm arbeitete, 1993 in seiner Dissertation *„Methods for Realistic Landscape Imaging“* [2] auf und zeigte Wege mit Fraktalen virtuelle Landschaften zu generieren. Als Ergebnis dieser und weiterer Forschungen entwickelte er ein Programm, welches die von ihm vorgestellten Techniken und Methoden nutzt. Dieses Programm wurde und wird von Pandromeda kommerziell unter dem Namen MojoWorld weiterentwickelt.

Es gibt heute eine ganze Reihe von Programmen die sich dieser Aufgabe widmen. Zu den bekanntesten zählen neben MojoWorld Terragen [4] und Bryce [5], die aber beide fotorealistische Landschaften mit Hilfe von Raytracing erzeugen. Dies ist aber in Echtzeit (noch) nicht möglich. Mehr dazu in Abschnitt 6.2.

Die menschliche Unterhaltung und Computerkunst bzw. computer art ist sicher eine der wichtigsten Anwendungen für virtuelle Landschaften. Sie strahlen oft einen eigentümlich, manchmal auch fantastischen Reiz aus, da sie im Gegensatz zu realen Landschaften nicht den natürlichen Bedingungen unterliegen (müssen), sondern auch Formen bilden können, die in der Natur oder im realen Leben kein Gegenstück haben. Auch in Computerspielen sind virtuelle Landschaften und Planeten von enormer Bedeutung, da man dem Spieler Vielfältigkeit und Abwechslungsreichtum bieten möchte. Einfache, nicht zu rechenintensive Algorithmen zur Formung von Gelände und ganzen Planeten spielen hier eine große Rolle.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit ist es, ein Verfahren zur Modellierung virtueller Planeten mit Hilfe von Fraktalen zu entwickeln. Dabei ist darauf zu achten, dass durch die Kugelform keine sphärischen Verzerrungen entstehen, die das Aussehen der Landschaft bzw. des Planeten negativ beeinflussen. Auf der Planetenoberfläche sollen dann mit fraktalen Algorithmen Gebirgs- und Höhenzüge gebildet werden, um so eine natürlich erscheinende Landschaft entstehen zu lassen. Wichtig ist dabei, dass sich das ganze in Echtzeit darstellen lässt, was zwei neue Probleme ins Spiel bringt. Erstens muss eine Art Level-Of-Detail (LOD) entwickelt werden, um nicht benötigte (weil nicht sichtbare) Teile des Planeten von der Berechnung und damit von der Darstellung auszuschließen bzw. den Detailgrad der Darstellung niedrig zu halten, wenn der gesamte Planet sichtbar ist. Dies impliziert ein weiteres Problem - die Persistenz des Planeten. Die meisten Algorithmen zur Erzeugung solcher Landschaften (siehe 3.2) arbeiten mit zufallsbasierten Höhenwerten. Dies hat zur Folge, dass einmal berechnete Werte für einen Punkt gespeichert werden und dann wieder geladen werden müßten, sollte dieser Punkt später noch einmal dargestellt werden. Anderfalls würde sich der Höhenwert bei einer erneuten Darstellung verändern, da er ja jedes mal zufällig berechnet wird. Das bedeutet für den Betrachter, dass er, wenn er einen bestimmten Punkt zu zwei verschiedenen Zeitpunkten betrachtet, zwei völlig verschiedene Landschaften zu sehen bekommen kann[1].

1.3 Gliederung

Im zweiten Abschnitt werden einige Grundlagen betrachtet, die eher geometrischer Natur sind. So werden einige der platonischen Körper und ihre Eigenschaften untersucht und es wird gezeigt, wie man man aus den Kenngrößen dieser Köper Koordinaten für den dreidimensionalen Raum errechnen kann.

In Abschnitt 3 werden kurz einige bekannte LOD-Algorithmen vorgestellt, und es wird gezeigt, warum diese hier nicht eingesetzt werden können. Im Anschluss daran wird eine Einführung in die wichtigsten Verfahren der Geländemodellie-

[1] Nämlich genau dann, wenn der Punkt auf Grund des LOD zwischendurch von der Darstellung ausgeschlossen wurde und sein Höhenwert verloren ging.

rung mit Hilfe von Fraktalen gegeben, von denen eines auch im neu entwickelten Algorithmus eingesetzt wird.

Danach wird in Abschnitt 4 das Verfahren vorgestellt, das im Rahmen dieser Diplomarbeit entwickelt wurde. Zuerst erfolgt eine kurze Betrachtung, wie man aus den im Kapitel 1 vorgestellten platonischen Körpern brauchbare Kugeldarstellungen für Boundery Representation erzeugen kann. Danach werden die einzelnen Schritte des Verfahrens eingehend besprochen und abschließend die Referenzimplementierung vorgestellt und erläutert.

Im vorletzten Abschnitt wird das Verfahren einer kritischen Betrachtung unterzogen. Es werden die Vor- und Nachteile der einzelnen Teile betrachtet und es werden einige Ansätze für Verbesserungen aufgezeigt.

Kapitel 6 schließt die Diplomarbeit mit einer Zusammenfassung ab und gibt einen kleinen Ausblick auf die zukünftige Entwicklung.

Kapitel 2

Platonische Körper

In diesem Kapitel werden kurz einige der Platonischen Körper vorgestellt, gezeigt, wie sie sich bei gegebener Seitenlänge im dreidimensionalen Raum konstruieren lassen.

Die platonischen Körper sind eine Gruppe von besonders regelmäßigen Polyedern mit einer Reihe von interessanten Eigenschaften. Ihre Seitenflächen bestehen aus regelmäßigen, zueinander kongruenten Vielecken. An ihren Ecken treffen immer gleichviele Seitenflächen aufeinander. Aus diesem Grund weisen sie eine hohe Symmetrie auf. Es gibt insgesamt 5 platonische Körper:

Name	Flächen	Ecken	Kanten
Tetraeder	4 gleichseitige Dreiecke	4	6
Hexaeder	6 Quadrate	8	12
Oktaeder	8 gleichseitige Dreiecke	6	12
Dodekaeder	12 reguläre Fünfecke	20	30
Ikosaeder	20 gleichseitige Dreiecke	12	30

Tabelle 2.1: Eigenschaften Platonischer Körper

Im folgenden werden Tetraeder, Oktaeder und Ikosaeder genauer betrachtet. Deren Flächen bestehen aus gleichseitigen Dreiecken, was eine wichtige Grundlage für das später vorgestellte Verfahren bildet. Es wird gezeigt, wie alle wichtigen Größes des jeweiligen Körpers berechnet werden. Insbesondere für den Ikosaeder wird zusätzlich geschildert, wie man aus diesen Größen Koordinaten für die Eckpunkte in einem dreidimensionalen Raum bestimmt.

2.1 Tetraeder

Ein Tetraeder ist ein platonischer Körper, der aus 4 gleichseitigen Dreiecken, mit Seitenlänge a, besteht. Daher folgt, dass die drei Innenwinkel dieser Dreiecke ebenfalls die gleiche Größe haben. Bei einer Innenwinkelsumme von 180 ° ergibt sich als Größe für einen Innenwinkel $\alpha = 60$ °. Nun lässt sich mit Hilfe des Satzes von Pythagoras die Höhe eines gleichseitigen Dreiecks herleiten:

$$a^2 = \left(\frac{a}{2}\right)^2 + h_D^2 \Rightarrow$$
$$h_D = \sqrt{a^2 - \left(\frac{a}{2}\right)^2} = \sqrt{2^2\left(\frac{a}{2}\right)^2 - \left(\frac{a}{2}\right)^2} = \sqrt{\left(\frac{a}{2}\right)^2 (2^2 - 1)} = \frac{a}{2}\sqrt{3}$$

Mit der Höhe eines gleichseitigen Dreiecks lässt sich nun auch dessen Schwerpunkt, als Schnittpunkt der Winkelhalbierenden, berechnen:

$$\tan\frac{\alpha}{2} = \frac{s_D}{\frac{a}{2}} \Rightarrow s_D = \frac{a}{2}\tan\frac{\alpha}{2} = \frac{h_D}{\sqrt{3}}\tan\frac{\alpha}{2} = \frac{h_D}{\sqrt{3}}\sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}$$

mit $\alpha = 60$ ° und $\cos\ 60° = \frac{1}{2}$ gilt nun:

$$s_D = \frac{h_D}{\sqrt{3}}\sqrt{\frac{1-\frac{1}{2}}{1+\frac{1}{2}}} = \frac{h_D}{\sqrt{3}}\sqrt{\frac{1}{3}} = \frac{h_D}{3}$$

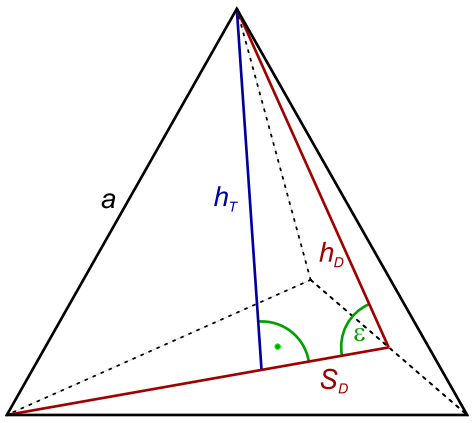

Abbildung 2.1: Tetraeder

Aus diesen Gleichungen kann nun die Höhe eines Tetraeders mit Seitenlänge

a berechnet werden:

$$a^2 = \left(\frac{2}{3}h_D\right)^2 + h_T^2 \Rightarrow$$

$$h_T = \sqrt{a^2 - \left(\frac{2}{3}h_D\right)^2} = \sqrt{a^2 - \left(\frac{2}{3}\frac{a}{2}\sqrt{3}\right)^2} = \sqrt{a^2 - \frac{3a^2}{9}} = \frac{\sqrt{6}}{3}a$$

Die Berechnung des Mittelpunktes des Tetraeders gestaltet sich etwas komplizierter als beim gleichseitigen Dreieck. Die Seitenhalbierenden zweier Seiten spannen zusammen mit der gegenüberliegenden Kante ein Dreieck auf. Dabei gilt für Winkel ε:

$$\cos\varepsilon = \frac{s_D}{h_D} = \frac{\frac{h_D}{3}}{h_D} = \frac{1}{3}$$

Und damit:

$$\tan\frac{\varepsilon}{2} = \sqrt{\frac{1-\cos\varepsilon}{1+\cos\varepsilon}} = \sqrt{\frac{1-\frac{1}{3}}{1+\frac{1}{3}}} = \sqrt{\frac{2}{4}} = \frac{\sqrt{2}}{2}$$

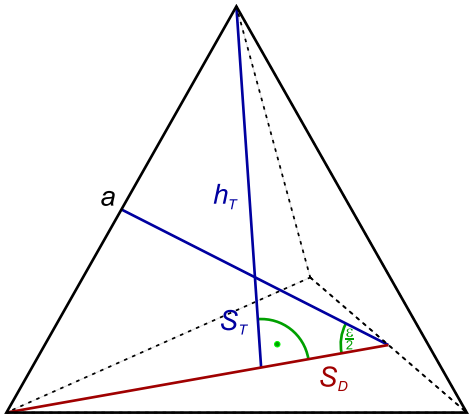

Abbildung 2.2: Schwerpunkt eines Tetraeders

Nun kann auch der Mittelpunkt des Tetraeders als Schnittpunkt zweier Raumhöhen ermitteln werden:

$$\tan\frac{\varepsilon}{2} = \frac{s_T}{s_D} = \frac{s_T}{\frac{h_D}{3}} = \frac{\sqrt{2}}{2} \Rightarrow s_T = \frac{\sqrt{2}h_D}{6} = \frac{\sqrt{2}\sqrt{3}\frac{a}{2}}{6} = \frac{\sqrt{6}}{12}a$$

Mit Hilfe dieser Formeln lässt sich leicht ein Tetraeder im dreidimensionalen Raum[1], mit dem Mittelpunkt im Koordinatenursprung, über seine Eckpunkte

[1]Wir gehen von einem dreidimensionalen Raum mit rechtshändigem, karthesischem Koordinatensystem aus.

A, B, C und D definieren. Der Einfachheit halber nehmen wir an, dass der Punkt A auf der positiven y-Achse liegt. Alle übrigen Anordnungen bzw. Orientierungen von Tetraedern, mit Mittelpunkt im Koordinatenursprung, lassen sich leicht durch Rotation um die Hauptachsen erreichen. Damit ergibt sich für die x- und z-Koordinate des Punktes A ein Wert von 0. Da der Abstand eines Eckpunktes des Tetraeders vom Mittelpunkt $h_T - s_T$ entspricht, ist $A = \left(0, \frac{\sqrt{6}}{4}a, 0\right)$.

Soll der Abstand aller Eckpunkte (d.h. der Radius der Umkugel des Tetraeders) 1 sein, so ergibt sich für die Seitenlänge:

$$\frac{\sqrt{6}}{4}a = 1 \Rightarrow a = \frac{4}{\sqrt{6}} \approx 1,633$$

Für die Höhe eines Seitendreiecks gilt dann:

$$h_D = \frac{a}{2}\sqrt{3} = \frac{\frac{4}{\sqrt{6}}}{2}\sqrt{3} = \frac{2\sqrt{3}}{\sqrt{6}} = \frac{2}{\sqrt{2}} \approx 1,414$$

Damit lassen sich auch die restlichen Eckpunkte B, C und D bestimmen.

$$A = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 \\ -\frac{1}{3} \\ \frac{2\sqrt{2}}{3} \end{pmatrix}$$

$$C = \begin{pmatrix} -\frac{2}{\sqrt{6}} \\ -\frac{1}{3} \\ -\frac{\sqrt{2}}{3} \end{pmatrix} \quad D = \begin{pmatrix} \frac{2}{\sqrt{6}} \\ -\frac{1}{3} \\ -\frac{\sqrt{2}}{3} \end{pmatrix}$$

Tabelle 2.2: Tetraederkoordinaten

2.2 Oktaeder

Ein weiterer platonischer Körper ist der Oktaeder, dessen Oberfläche von 8 gleichseitigen Dreiecken gebildet wird. Man kann ihn auch als Doppelpyramide mit gleich langen Kanten ansehen. Für eine dieser Pyramiden, mit Seitenlänge a, ist die Länge der Diagonale des Basisquadrats:

$$d = \sqrt{a^2 + a^2} = a\sqrt{2}$$

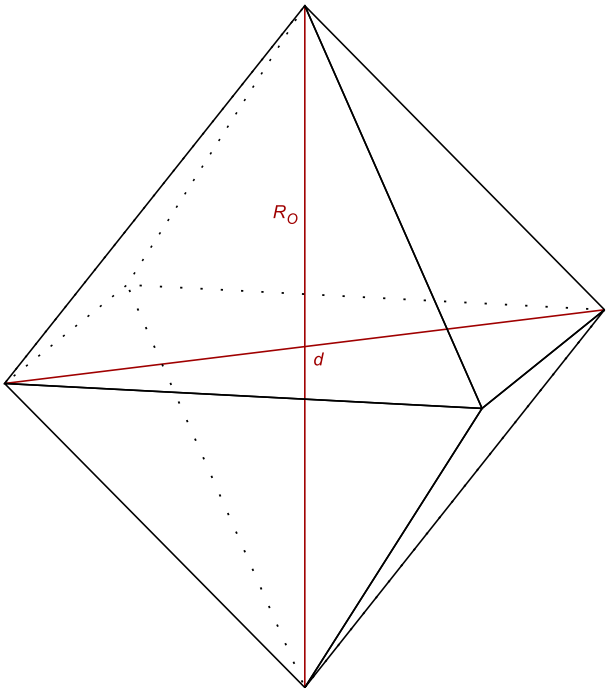

Abbildung 2.3: Höhe eines Oktaeder

Damit ergibt sich für die Höhe dieser Pyramide (die gleichzeitig der Radius der Umkugel des Oktaeders ist):

$$a^2 = \left(\frac{d}{2}\right)^2 + R_O^2 \Rightarrow$$

$$R_O = \sqrt{a^2 - \left(\frac{d}{2}\right)^2} = \sqrt{a^2 - \left(\frac{a\sqrt{2}}{2}\right)} = \sqrt{a^2 - \frac{a^2}{2}} = \sqrt{\frac{a^2}{2}} = \frac{a}{2}\sqrt{2}$$

Dies entspricht natürlich $\frac{d}{2}$, da aus Symmetriegründen die Höhe dieser Pyramide der halben Diagonale einer Pyramide entspricht, die um 90 ° um die durch d festgelegte Achse gedreht wurde.

Der Mittelpunkt des Oktaeders liegt genau im Schnittpunkt aller Diagonalen des Oktaeders bzw. dem Schnittpunkt der Diagonalen der Basis einer Pyramide. Möchte man einen Oktaeder mit einem Umkugelradius von 1, so gilt:

$$R_O = \frac{a}{2}\sqrt{2} = 1 \Rightarrow a = \frac{2}{\sqrt{2}} \approx 1,414$$

Ein solcher Oktaeder, bei dem 4 Eckpunkte in der xz-Ebene und 2 jeweils auf der positiven bzw. negativen y-Achse liegen, lässt sich leicht über seine Koordi-

naten definieren. Wie beim Tetraeder lassen sich andere Orientierungen einfach über Rotationen erreichen.

$$A = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad B = \begin{pmatrix} -\frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \quad C = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$D = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \quad E = \begin{pmatrix} -\frac{1}{\sqrt{2}} \\ 0 \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \quad F = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$$

Tabelle 2.3: Oktaederkoordinaten

2.3 Ikosaeder

Ein Ikosaeder hat 20 gleichseitige, kongruente Dreiecke als Flächen. Legt man durch zwei gegenüberliegenden Kanten einen Schnitt durch den Ikosaeder, erhält man ein unregelmäßiges Sechseck, bestehend aus den beiden Kanten und aus den Seitenhalbierenden der vier angrenzenden Dreiecke. Abbildung 2.4 zeigt einen solchen Schnitt und verschiedene, im Folgenden benutzte Kenngrößen des Ikosaeders.

Die Länge der Seitenhalbierenden (Höhe) eines gleichseitigen Dreiecks ist, wie in Abschnitt 2.1 gezeigt,

$$h_D = \frac{a}{2}\sqrt{3}$$

und es gilt aus Symmetriegründen $k = l$ und $m = k - \frac{a}{2}$. Damit ist $h_D^2 = k^2 + (k - \frac{a}{2})^2$ und es gilt die Gleichung

$$\frac{3}{4}a^2 = k^2 + \left(k - \frac{a}{2}\right)^2 = 2k^2 - ak + \frac{a^2}{4} \Rightarrow k^2 - \frac{a}{2}k = \frac{a^2}{4}$$
$$\Rightarrow k^2 - \frac{a}{2}k - \frac{a^2}{4} = 0$$

deren positive Lösung $k = \frac{a}{4}(\sqrt{5}+1)$ ist. Daraus ergibt sich $m = \frac{a}{4}(\sqrt{5}+1) - \frac{a}{2} = \frac{a}{4}(\sqrt{5}-1)$. Jetzt lässt sich R_I als Radius der Umkugel des Ikosaeders berechnen,

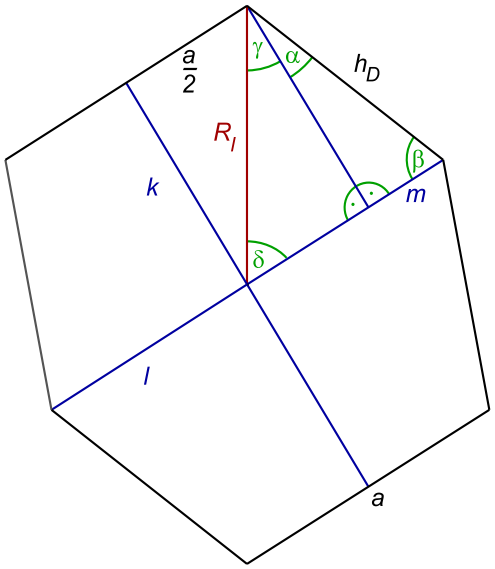

Abbildung 2.4: Ikosaederschnitt A

denn

$$R_I = \sqrt{k^2 + \frac{a^2}{4}} = \sqrt{\left(\frac{a}{4}(\sqrt{5}+1)\right)^2 + \frac{a^2}{4}} = \sqrt{\frac{a^2}{8}\left(\sqrt{5}+3\right) + \frac{a^2}{4}}$$
$$= \sqrt{\frac{a^2}{8}\left(\sqrt{5}+5\right)} = \frac{a}{4}\sqrt{10+2\sqrt{5}}$$

und für einen Ikosaeder mit einem Umkugelradius $R_I = 1$ gilt somit die Seitenlänge seiner Dreiecke $a = \frac{4}{\sqrt{10+2\sqrt{5}}} \approx 1,051$ und ihre Höhe $h_D = \frac{2\sqrt{3}}{\sqrt{10+2\sqrt{5}}} \approx 0,911$.

Für die Berechnung der Koordinaten der Eckpunkte ist auch die Größe der Innenwinkel des Ikosaeders von Bedeutung. Sie lassen sich aber leicht mit Hilfe der trigonometrischen Funktionen ermitteln.

$$\sin\alpha = \frac{m}{h_D} = \frac{\frac{a}{4}\left(\sqrt{5}-1\right)}{\frac{a}{2}\sqrt{3}} = \frac{\sqrt{5}-1}{2\sqrt{3}} \Rightarrow \alpha \approx 20,905°$$

$$\sin\beta = \frac{k}{h_D} = \frac{\frac{a}{4}\left(\sqrt{5}+1\right)}{\frac{a}{2}\sqrt{3}} = \frac{\sqrt{5}+1}{2\sqrt{3}} \Rightarrow \beta \approx 69,0945°$$

$$\sin\gamma = \frac{\frac{a}{2}}{R_I} = \frac{\frac{a}{2}}{\frac{a}{4}\sqrt{10+2\sqrt{5}}} = \frac{2}{\sqrt{10+2\sqrt{5}}} \Rightarrow \gamma \approx 31,717°$$

$$\sin\delta = \frac{k}{R_I} = \frac{\frac{a}{4}\left(\sqrt{5}+1\right)}{\frac{a}{4}\sqrt{10+2\sqrt{5}}} = \frac{\sqrt{5}+1}{\sqrt{10+2\sqrt{5}}} \Rightarrow \delta \approx 58,283°$$

Die Berechnung der Koordinaten der Eckpunkte eines solchen Ikosaeders aus dem Radius seiner Umkugel gestaltet sich schwieriger als beim Tetraeder oder Oktaeder und wird deshalb hier gesondert aufgeführt. Es werden lediglich die Koordinaten eines Quadranten berechnet, der Rest lässt sich aufgrund der hohen Symmetrieeigenschaften leicht durch vertauschen der Vorzeichen einzelner Koordinaten erschließen. Wie schon beim Tetraeder und beim Oktaeder legen wir die Koordinaten für einen Punkt fest. Sei $A = (0, 1, 0)$, damit ist sein gegenüberliegender Punkt L ebenfalls mit $L = (0, -1, 0)$ definiert. Da die xy-Ebene das Dreieck ACF „halbiert“, gilt für die z-Koordinate des Punktes C $z_C = \frac{a}{2} = \frac{2}{\sqrt{10+2\sqrt{5}}}$. Die x-Koordinate von C ist

$$x_C = \sin(\alpha + \gamma)\, h_D = (\sin\alpha\cos\gamma + \cos\alpha\sin\gamma)\, h_D$$

und lässt sich aus den zuvorberechneten Winkeln[2] und der Dreieckshöhe h_D leicht berechnen:

$$\begin{aligned} x_C &= \left(\frac{\sqrt{5}-1}{2\sqrt{3}} \cdot \frac{\sqrt{5}+1}{\sqrt{10+2\sqrt{5}}} + \frac{\sqrt{5}+1}{2\sqrt{3}} \cdot \frac{2}{\sqrt{10+2\sqrt{5}}}\right) \frac{2\sqrt{3}}{\sqrt{10+2\sqrt{5}}} \\ &= \frac{2\sqrt{5}+6}{10+2\sqrt{5}} = \frac{\sqrt{5}+3}{\sqrt{5}+5} \end{aligned}$$

y_C kann nun mit y_A, x_C und h_D errechnet werden:

$$\begin{aligned} y_C &= y_A - \sqrt{h_D^2 - x_C^2} = 1 - \sqrt{\left(\frac{2\sqrt{3}}{\sqrt{10+2\sqrt{5}}}\right)^2 - \left(\frac{\sqrt{5}+3}{\sqrt{5}+5}\right)^2} \\ &= 1 - \sqrt{\frac{16}{\left(\sqrt{5}+5\right)^2}} = \frac{\sqrt{5}+1}{\sqrt{5}+5} \\ &\Rightarrow C = \left(\frac{\sqrt{5}+3}{\sqrt{5}+5}, \frac{\sqrt{5}+1}{\sqrt{5}+5}, \frac{2}{\sqrt{10+2\sqrt{5}}}\right) \end{aligned}$$

Zur Berechnung von Punkt E genügen der Punkt L, $\sin\gamma$ und $\sin\delta$. $z_E = 0$, da der Punkt E genau wie die Punkte A und L auf der xy-Ebene liegt.

$$x_E = \sin\delta \cdot a = \frac{\sqrt{5}+1}{\sqrt{10+2\sqrt{5}}} \frac{4}{\sqrt{10+2\sqrt{5}}} = \frac{2\sqrt{5}+2}{\sqrt{5}+5}$$

[2]Im rechtwinkligen Dreieck ist $\cos\alpha = \sin\beta$.

$$y_E = y_L + \sin\gamma \cdot a = \frac{2}{\sqrt{10+2\sqrt{5}}}\frac{4}{\sqrt{10+2\sqrt{5}}} - 1 = \frac{8}{\sqrt{10+2\sqrt{5}}} - 1 = -\frac{\sqrt{5}+1}{\sqrt{5}+5}$$

$$\Rightarrow E = \left(\frac{2\sqrt{5}+2}{\sqrt{5}+5}, -\frac{\sqrt{5}+1}{\sqrt{5}+5}, 0\right)$$

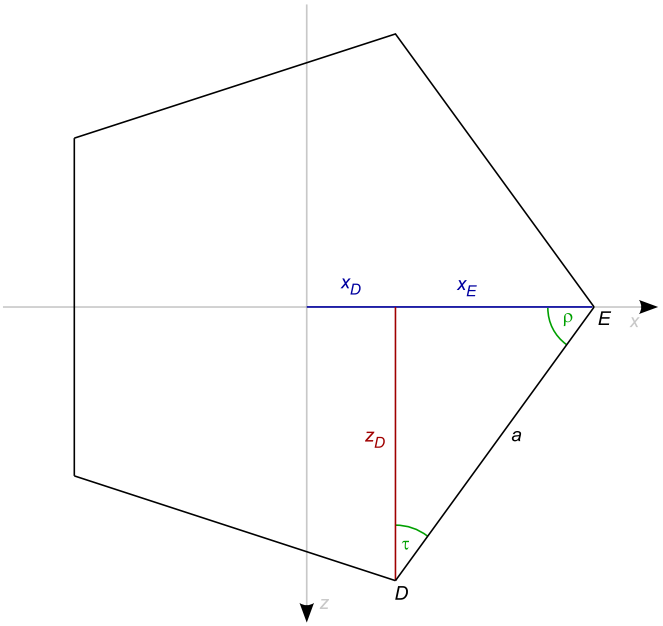

Abbildung 2.5: Ikosaederschnitt B

Punkt D ist nicht so einfach berechnen, wie die vorhergehenden, denn keine seiner Kanten läuft parallel zu einer der Koordinatenachsen. Legt man aber einen Schnitt parallel zur xz-Ebene durch den Punkt D (siehe dazu Abbildung 2.5), so erhält man an der Schnittfläche ein regelmäßiges Pentagon. Die Größe der Innenwinkel beträgt 108 °. Mit Hilfe dieser Winkel[3] und den Koordinaten des Punktes E errechnen sich die Koordinaten von D wie folgt:

$$x_D = x_E - \sin\tau \cdot a = \frac{2\sqrt{5}+2}{\sqrt{5}+5} - \frac{4\sin\left(\frac{\pi}{5}\right)}{\sqrt{10+2\sqrt{5}}}$$

$$y_D = y_E$$

[3] $\rho = 54\,°$ und $\tau = 36\,°$

$$z_D = \sin\rho \cdot a = \sin\left(\frac{3\pi}{10}\right)\frac{4}{\sqrt{10+2\sqrt{5}}} = \frac{4\sin\left(\frac{3\pi}{10}\right)}{\sqrt{10+2\sqrt{5}}}$$

Alle weiteren Punkte des Ikosaeders sind Spiegelungen der Punkte C, D und E an der xy-, xz- oder zy-Ebene des Koordinatensystems. So entspricht Punkt F Punkt C mit negativer z-Koordinate. Tabelle 2.4 gibt die auf diese Weise errechneten Koordinaten an.

Die errechneten Werte lassen sich leicht auf ihre Richtigkeit überprüfen. Für jeden Punkt P gilt $\sqrt{x_P^2+y_P^2+z_P^2}=1$ (die Entfernung des Punktes zum Koordinatenursprung ist 1) und für je zwei Punkte P und Q, die zu einem Dreieck des Ikoseaders gehören, ist $\sqrt{(x_P-x_Q)^2+(y_P-y_Q)^2+(z_P-z_Q)^2}=a$ die Länge der Seitendreiecke.

$$A=\begin{pmatrix}0\\1\\0\end{pmatrix} \qquad B=\begin{pmatrix}-\frac{2\sqrt{5}+2}{\sqrt{5}+5}+\frac{4\sin\left(\frac{\pi}{5}\right)}{\sqrt{10+2\sqrt{5}}}\\ \frac{\sqrt{5}+1}{\sqrt{5}+5}\\ \frac{4\sin\left(\frac{3\pi}{10}\right)}{\sqrt{10+2\sqrt{5}}}\end{pmatrix} \qquad C=\begin{pmatrix}\frac{\sqrt{5}+3}{\sqrt{5}+5}\\ \frac{\sqrt{5}+1}{\sqrt{5}+5}\\ \frac{2}{\sqrt{10+2\sqrt{5}}}\end{pmatrix}$$

$$D=\begin{pmatrix}\frac{2\sqrt{5}+2}{\sqrt{5}+5}-\frac{4\sin\left(\frac{\pi}{5}\right)}{\sqrt{10+2\sqrt{5}}}\\ -\frac{\sqrt{5}+1}{\sqrt{5}+5}\\ \frac{4\sin\left(\frac{3\pi}{10}\right)}{\sqrt{10+2\sqrt{5}}}\end{pmatrix} \qquad E=\begin{pmatrix}\frac{2\sqrt{5}+2}{\sqrt{5}+5}\\ -\frac{\sqrt{5}+1}{\sqrt{5}+5}\\ 0\end{pmatrix} \qquad F=\begin{pmatrix}\frac{\sqrt{5}+3}{\sqrt{5}+5}\\ \frac{\sqrt{5}+1}{\sqrt{5}+5}\\ -\frac{2}{\sqrt{10+2\sqrt{5}}}\end{pmatrix}$$

$$G=\begin{pmatrix}\frac{2\sqrt{5}+2}{\sqrt{5}+5}-\frac{4\sin\left(\frac{\pi}{5}\right)}{\sqrt{10+2\sqrt{5}}}\\ -\frac{\sqrt{5}+1}{\sqrt{5}+5}\\ -\frac{4\sin\left(\frac{3\pi}{10}\right)}{\sqrt{10+2\sqrt{5}}}\end{pmatrix} \qquad H=\begin{pmatrix}-\frac{2\sqrt{5}+2}{\sqrt{5}+5}+\frac{4\sin\left(\frac{\pi}{5}\right)}{\sqrt{10+2\sqrt{5}}}\\ \frac{\sqrt{5}+1}{\sqrt{5}+5}\\ -\frac{4\sin\left(\frac{3\pi}{10}\right)}{\sqrt{10+2\sqrt{5}}}\end{pmatrix} \qquad I=\begin{pmatrix}-\frac{\sqrt{5}+3}{\sqrt{5}+5}\\ -\frac{\sqrt{5}+1}{\sqrt{5}+5}\\ -\frac{2}{\sqrt{10+2\sqrt{5}}}\end{pmatrix}$$

$$J=\begin{pmatrix}-\frac{2\sqrt{5}+2}{\sqrt{5}+5}\\ \frac{\sqrt{5}+1}{\sqrt{5}+5}\\ 0\end{pmatrix} \qquad K=\begin{pmatrix}-\frac{\sqrt{5}+3}{\sqrt{5}+5}\\ -\frac{\sqrt{5}+1}{\sqrt{5}+5}\\ \frac{2}{\sqrt{10+2\sqrt{5}}}\end{pmatrix} \qquad L=\begin{pmatrix}0\\-1\\0\end{pmatrix}$$

Tabelle 2.4: Ikosaederkoordinaten

Kapitel 3

LOD-Algorithmen und Fraktale Landschaften

Um ganze Planeten in einem hohen Detailgrad darzustellen, werden mehr Polygone benötigt, als der derzeitige Stand der Technik zulässt. Aus diesem Grund ist es notwendig, nicht benötigte (weil nicht sicht- bzw. kaum unterscheidbar) Teile der Welt von der Berechnung auszuschließen oder zumindest nur in einer groben Detailstufe darzustellen, um Polygone einzusparen. Einige bekannte Verfahren, Level-Of-Detail-Algorithmen oder kurz LOD-Algorithmen, werden im ersten Teil dieses Kapitels vorgestellt. Der zweite Teil beschäftigt sich mit Algorithmen, die mit Hilfe von Fraktalen gebirgsähnliche Strukturen erzeugen. Diese bilden die Grundlage für das in dieser Diplomarbeit vorgestellte Verfahren.

3.1 LOD-Algorithmen

Betrachtet man eine typische Szene, so fällt auf, dass bei weit entfernten oder sehr kleinen Objekten Details kaum oder gar nicht zu erkennen sind. Dennoch müssen alle Polygone dieses Objekts die Rendering-Pipeline durchlaufen. Die Polygonanzahl ist aber immer noch ein limitierendes Element in der Darstellung virtueller Szenen. Die Verarbeitung von Polygonen, die nicht oder kaum sichtbar sind, ist somit eine Verschwendung von Ressourcen, die an anderer Stelle sinnvoller eingesetzt werden könnten. Deshalb wurden schnell Techniken entwickelt, die sich dieses Problems annahmen. Eine kurze Einführung in dieses Thema bietet die Studienarbeit von Stefan Rahn [7].

Die erste Veröffentlichung zu diesem Thema von James Clark [8] schlug vor, alle Objekte einer Szene in verschiedenen Auflösungsstufen zu modellieren und dann je nach Bedarf zwischen den einzelnen Auflösungen hin und her zu schalten. Dabei werden die geringeren Auflösungen eines Modells entweder komplett neu modelliert oder durch Vereinfachung der höchsten Auflösung gewonnen. Dies wird in einem seperaten Schritt vor der eigentlichen Darstellung[1] vorgenommen und verursacht somit keinen wesentlichen Mehraufwand. Abbildung 3.1 zeigt das Bunny aus dem Stanford 3D Scanning Repository [9] in 3 verschiedenen Auflösungen.

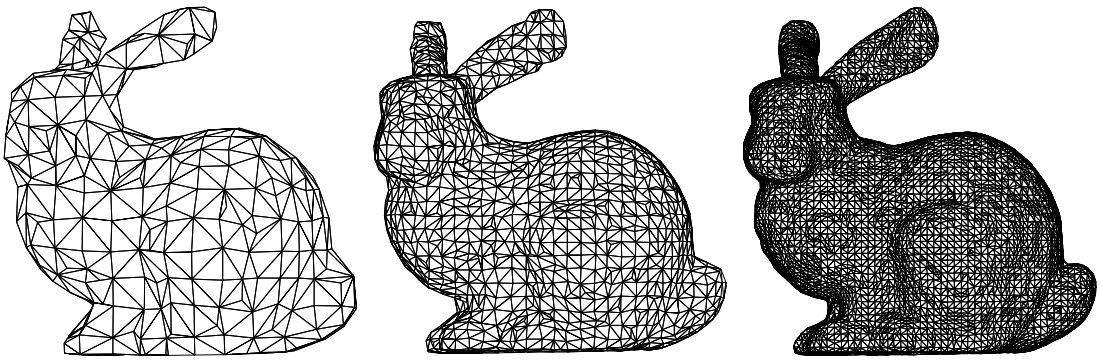

Abbildung 3.1: Auflösungen bei diskreten LOD

Das Wechseln zwischen den einzelnen Auflösungen verursacht eine mehr oder weniger starke Veränderung des Bildes, die vom Betrachter im Allgemeinen als störend empfunden wird. Man könnte dieses, *popping* genannte, Problem verringern, indem man sehr viele Auflösungsstufen verwendet. Der Unterschied zwischen den einzelnen Modellen ist dann geringer, was auch das *popping* verringert. Allerdings ist des Vorgehen unpraktikabel, da es ebenfalls viele Ressourcen verschlingt.

Kontinuierliche LOD-Techniken versuchen dieses Problem zu vermeiden. Hier werden nicht mehrere Modelle zur Darstellung genutzt, sondern es wird von einem Orginalmodell mit hohem Detailgrad aus eine Datenstruktur (im Allgemeinen ein Baum) erstellt, die Vereinfachungsschritte speichert. Bei diesen Vereinfachungsschritten handelt es sich in der Regel um das Zusammenfassen von zwei oder mehr Dreiecken zu einem großen Dreieck. Im Gegensatz zu diskretem LOD kann sehr

[1] d. h. vor Programmablauf

viel feiner zwischen verschiedenen Detailstufen hin- und hergeschaltet werden, was das *popping*-Problem deutlich reduziert.

Aber auch diese Vorgehensweise hat einige Nachteile. Sie ist für die Darstellung eines ganzen Planeten nicht geeignet, da gesamte der Planet in voller Aulösung dargestellt werden müsste, falls sich der Betrachter direkt auf der Oberfläche befindet. An diesem Punkt setzen blickpunktabhängige LOD-Verfahren an, indem sie ein variables Level-of-Detail nutzen. Dazu setzen sie blickpunktabhängige Kriterien ein[2], um für den jeweiligen Blickpunkt den besten Detailgrad zu ermitteln. Auf diese Art und Weise lässt sich LOD auch für sehr große Objekte nutzen, da nur ein kleiner Teil in einem hohen Detailgrad dargestellt werden muss, während der überwiegende Rest mit wenigen Polygonen auskommt. Zur Landschaftsvisualisierung sind sie deshalb von zentraler Bedeutung. Die wichtigsten werden kurz vorgestellt.

3.1.1 Lindstrom

1996 wurde von Lindstrom et al. ein Algorithums vorgestellt, der ein regelmäßiges Gitter von Höhenwerten nutzt. Es wird von der detailiertesten Triangulierung ausgehend schrittweise durch Zusammenfassen von Dreiecken vereinfacht, bis ein bestimmtes Fehlermaß erreicht wird. Dieses Fehlermaß muss allerdings in einem *bottom-up*-Verfahren ermittelt werden und die Höhenwerte müssen für alle Punkte im Voraus vorliegen. Durch das Zusammenfassen der Dreiecke wird die Polygonanzahl deutlich verringert. Das festgelegte Fehlermaß sorgt für eine gute Bildqualität. Leider müssen dazu, wie bereits erwähnt, alle Höhenwerte vorliegen und geladen werden, was für ganze Planeten mit hohem Detailgrad nicht möglich ist. Außerdem skaliert der Algoritmus durch den *bottom-up*-Ansatz schlecht.

3.1.2 ROAM

Dieses Problem vermeidet der ROAM-Algorithmus (Real-time Optimally Adapting Meshes) [11] von Mark Duchaineau et al. von 1997, indem er ein *top-down*-Ansatz verwendet. Im Gegensatz zu Lindstrom und dem nachfolgend vorgestellten C-LOD-Algorithmus von Röttger werden hier die Dreiecke in einem *bin tree* verwaltet. Begonnen wird mit einem gleichschenkligen, rechtwinkligen

[2] z.B. die Entfernung zum Betrachter oder das View Frustum

Dreieck, welches die Wurzel des *bin trees* bildet. Dieses wird an der Basis in zwei neue, ebenfalls gleichschenklige rechwinklig aufgeteilt. Diese beiden neuen Dreiecke werden jetzt zu Kindern der Wurzel. Auch diese beiden Kinder werden, wie oben beschrieben, geteilt und erhalten somit ihrerseits Kinder. Dies kann beliebig lange rekursiv durchgeführt werden. Abbildung 3.2 verdeutlicht diesen Vorgang. Vom Startdreieck links oben ausgehend werden 5 Zerlegungsschritte für jeweils alle Dreiecke durchgeführt.

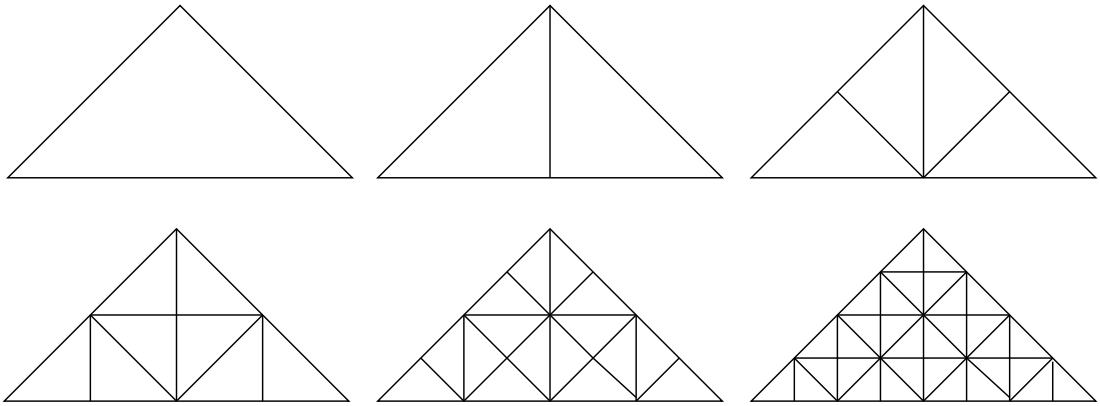

Abbildung 3.2: Dreieckszerlegung bei ROAM

Der ROAM-Algorithmus steuert nun, wo Dreiecke aufgeteilt und wieder zusammengefügt werden. Dazu nutzt ROAM zwei Prioritätswarteschlangen, die *split queue* und die *merge queue*, die die Teilungs- bzw. Vereinigungsoperationen enthalten und deren Priorität ein Fehlermaß ist. Um dieses Fehlermaß zu ermitteln, sind zwei Fehlermetriken notwendig. Die erste verwendet ein *bottom-up*-Verfahren, um den Unterschied zwischen zwei (zusammenzufassenden) Dreiecken und dem Vaterdreieck[3] zu bestimmen. Diese Fehlermessung kann vorab durchgeführt und zusammen mit dem Höhenfeld abgespeichert werden und dient praktisch als Parameter einer zweiten Fehlermetrik, die den eigentlichen Bildfehler nach der *viewing transformation* bestimmt. Die Prioriät der *split queue* entspricht dann dem Maß, wie stark eine Teilungsoperation einen Fehler reduziert. Diese sind zu bevorzugen, da man mit jeder Teilungsoperation die größtmögliche Reduzierung des Bildfehlers erreichen will. Wird eine zuvor festgelegte Schranke für den Bildfehler unterschritten, sind keine weiteren Teilungsoperationen notwendig. Die *merge queue* funktioniert ganz analog, indem sie Vereingungsoperationen bevor-

[3]dem Vater im *bin tree*

zugt, die möglichst geringe Bildfehler verursachen. Zwar ist es möglich, ROAM auch ohne *merge queue* zu implementieren, allerdings müsste dann der *bin tree* jedesmal komplett neu aufgebaut werden. Mit der *merge queue* kann der einmal erzeugte *bin tree* weiterverwendet werden, die Effiziens des Algorithmus lässt sich dadurch deutlich steigern.

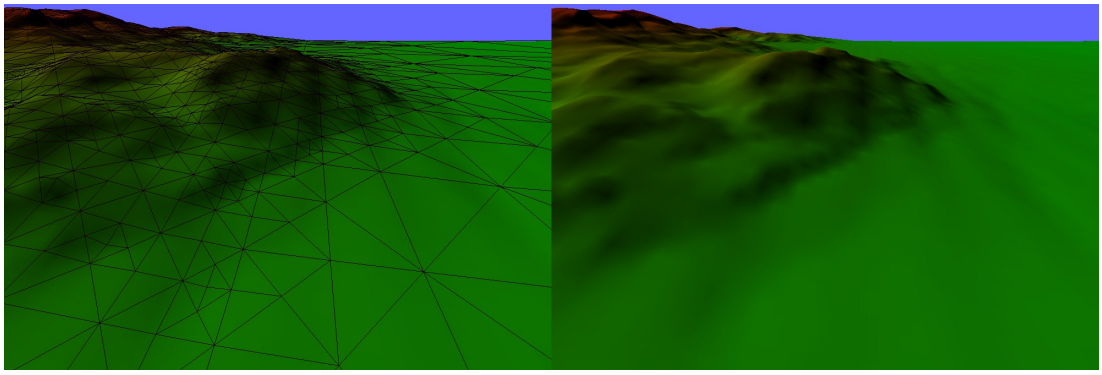

Abbildung 3.3: mit ROAM visualisiertes Terrain [11]

Abbildung 3.3 zeigt eine mit ROAM visualisierte Landschaft. Links mit, rechts ohne dem zu Grunde liegende Dreiecksgitter. Dieses wird in Abbildung 3.4 noch einmal genauer gezeigt. Das gelbe Quadrat kennzeichnet die Betrachterposition. Die in Abbildung 3.3 sichtbaren Dreiecke sind hell-, die teilweise sichtbaren Dreiecke sind dunkelgrün dargestellt. Blaue Dreieck in Abbildung 3.4 sind in Abbildung 3.3 nicht sichtbar.

Durch den *top-down*-Ansatz ist dieses Verfahren deutlich effizienter als Lindstrom, da hier die Berechnungen nur bis zur benötigten Auflösung und nicht für alle Detailstufen durchgeführt werden müssen. ROAM ist deshalb zur Zeit wahrscheinlich eines der am häufigsten eingesetzten Terrain-Rendering-Verfahren.

3.1.3 Röttger

Der von Röttger et al. entwickelte C-LOD-Algorithmus [12] basiert auf Lindstroms Algorithmus und benutzt wie dieser einen *quad tree*, behebt allerdings dessen Schwächen. So wird statt der *bottom-up*-Strategie wie bei ROAM ein *top-down*-Ansatz gewählt. Außerdem wird ein Geomorphing eingesetzt, um den *popping*-Effekt beim Einfügen neuer Dreiecke zu verringern. Wie weit ein Knoten im *quad tree* verfeinert wird, hängt von 2 verschiedenen Kriterien ab. Zum einen

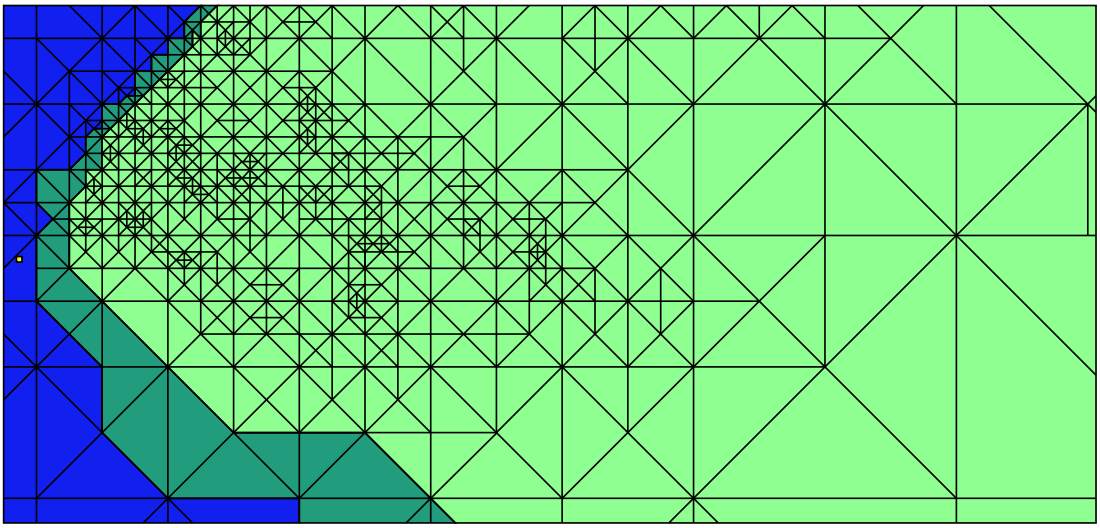

Abbildung 3.4: Dreiecksgitter zur Abbildung 3.3 (neu erstellt nach [11])

verringert sich die Auflösung des Dreiecksnetzes mit zunehmendem Abstand zum Betrachter. Dies erreicht Röttger, indem erfolgende Ungleichung auswertet wird:

$$\frac{l}{d} < C$$

Dabei ist l die Entfernung des Blockmittelpunkts zum Betrachter, d die Seitenlänge des Blocks und C eine Konstante. Die Blöcke des *quad tree* werden solange unterteilt, bis die Ungleichung nicht mehr erfüllt ist. C kann somit als Qualitäts-Parameter angesehen werden. Je höher C ist, desto kleiner die Blöcke im *quad tree* und um so höher die Auflösung des Geländes. Andererseits, je kleiner l ist, desto kleiner müssen die Blöcke werden, um die Ungleichung nicht mehr zu erfüllen. Das führt dazu, dass diejenigen Blöcke, die näher am Betrachter sind, automatisch weiter unterteilt werden als jene, welche weiter entfernt sind.

Als weiteres Kriterium dient die Oberflächenbeschaffenheit. Wird innerhalb des *quad tree* eine Ebene aufgestiegen und somit vier Blöcke zu einem zusammengefasst, werden fünf Vertizes entfernt. Der Fehler ergibt sich nun aus den Höhendifferenzen der zu entfernenden Vertizes zu den durch lineare Interpolation entstandenen Mittelpunkten der Seiten bzw. der Diagonalen des Blocks, wie Abbildung 3.5 zeigt. Aus dem Maximum der Höhenunterschiede dh_1 bis dh_6 gewichtet mit der Seitenlänge des Blocks ergibt sich nun seine Oberflächenbeschaffenheit.

$$d2 = \frac{1}{d} \max_{i=1..6} |dh_i|$$

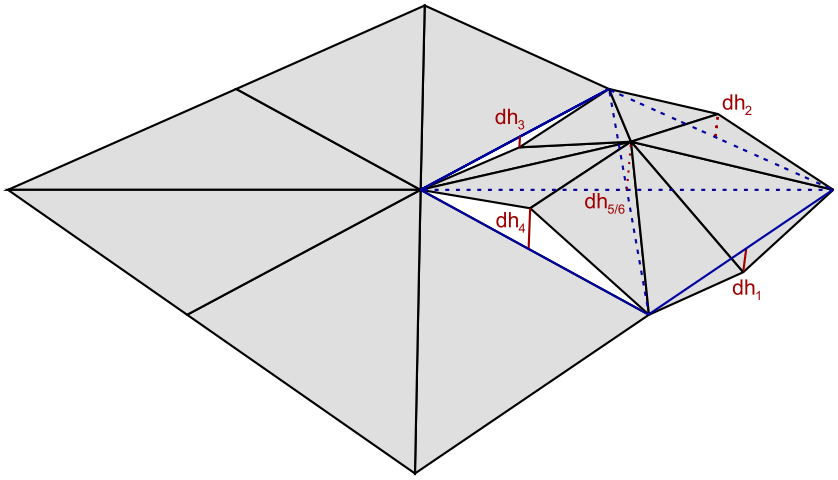

Abbildung 3.5: Fehlermetrik bei Röttger(neu erstellt nach[12])

Wie bei ROAM wird dies vorab in einer *bottom-up*-Analyse berechnet und im *quad tree* abgespeichert. Diese beiden Entscheidungskriterien werden durch folgende Gleichung zusammengefasst und ausgewertet:

$$f = \frac{1}{d \cdot C \cdot \max(c \cdot d2, 1)}$$

Zur Laufzeit wird dann der *quad tree top-down* traversiert und f für jedes Blatt ausgewertet. Ist $f \geq 1$, wird der Block verfeinert und f für seine Kinder geprüft. Falls $f < 1$ ist, wird der rekursive Abstieg im Baum an dieser Stelle gestoppt und der Block wird ausgegeben bzw. gezeichnet.

3.2 Fraktale Landschaften

Im folgenden werden einige Algorithmen zur fraktalen Modellierung von Gelände aufgezeigt und kurz beschrieben. Vorgestellt werden Perlin Noise, Plasma und Diamond-Square, da diese auch heute noch am verbreitetsten sind, obwohl sie bereits in den achtziger Jahren des letzten Jahrhunderts entwickelt wurden.

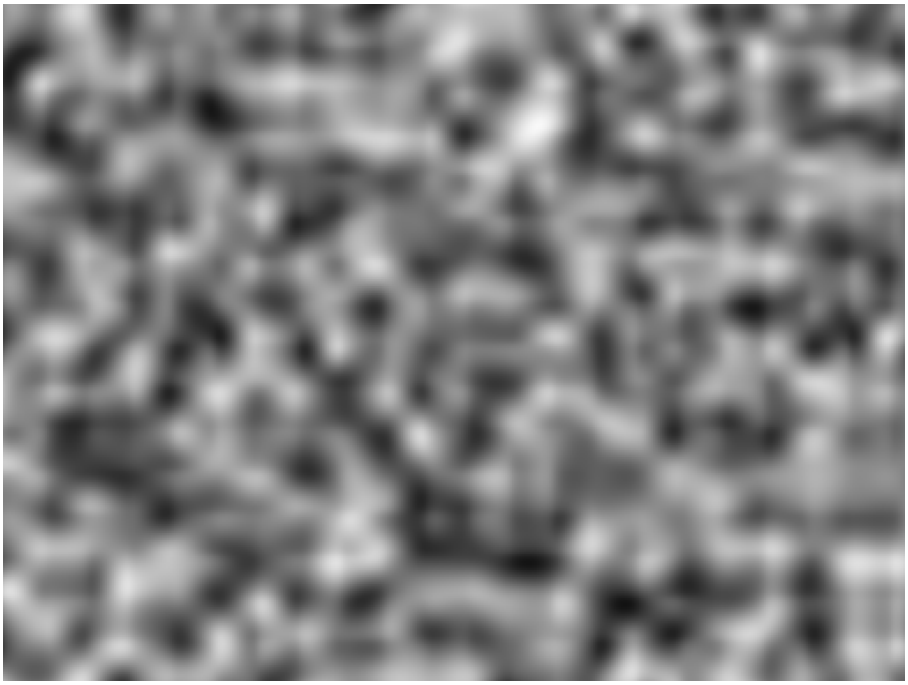

Abbildung 3.6: Perlin Noise

3.2.1 Perlin Noise

Perlin Noise ist ein 1983 von Ken Perlin entwickeltes Verfahren zur Erzeugung von koherentem Rauschen (Coherent Noise). Perlin hat das Verfahren ursprünglich entwickelt, um natürlich wirkende Texturen zu erstellen. Es lässt sich aber auch in vielen anderen Bereichen einsetzen, um natürlich wirkende Muster, wie Holzmaserung, Marmor, Wolken, Feuer oder Gebirgszüge zu erzeugen. Hier erfolgt nur eine kurze Einführung, das Verfahren wird ausführlich in [3] vorgestellt. Weitere Dartstellungen finden sich im WWW unter [13].

Rauschfunktion

Perlin Noise erzeugt man durch Aufaddieren verschiedener Rauschfunktionen, wobei man typischerweise eine Reihe von Funktionen hat, bei denen sich, ausgehend von einer Startfrequenz und -amplitude (aus praktischen Gründen üblicherweise Zweierpotenzen), die Frequenz jeweils verdoppelt, während sich die Amplitude halbiert. Abbildung 3.7 zeigt eine Gruppe solcher Funktionen und Abbildung 3.8 das Ergebnis ihrer Addition. Desweiteren benötigt man eine Interpolationsfunktion (siehe Abschnitt 3.2.1).

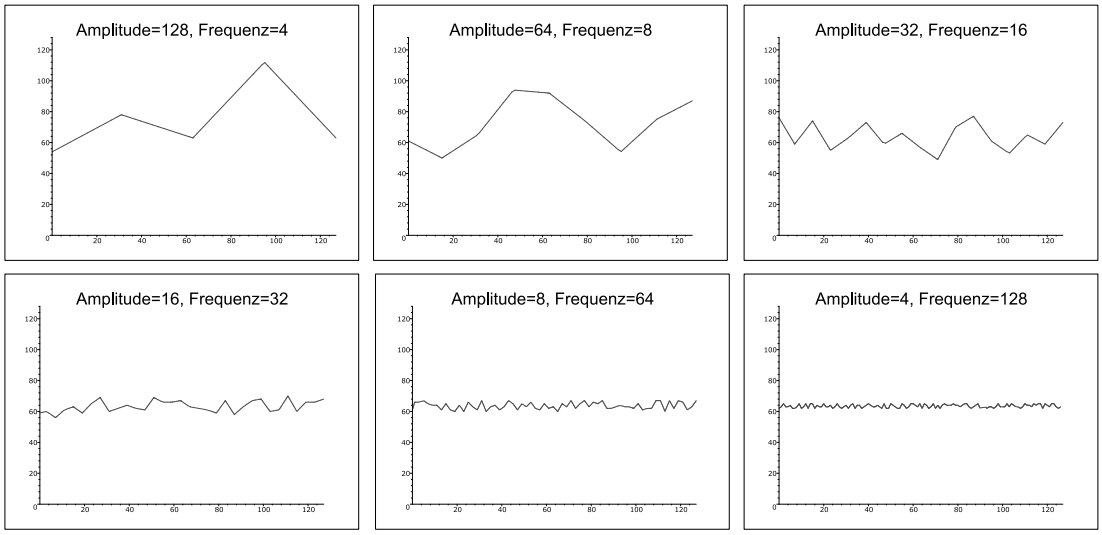

Abbildung 3.7: Rauschfunktionen

Eine Rauschfunktion selbst ist eine Funktion, die Pseudozufallszahlen erzeugt. Das heißt, es werden Zahlen erzeugt, die zufällig aussehen, tatsächlich aber durch die Parameter der Funktion deterministisch bestimmt sind. Gleiche Parameter liefern gleiche (Pseudo-)Zufallszahlen. Man kann hier keine echten Zufallswerte nehmen, da man bei einer erneuten Berechnung das gleiche Muster erhalten will. Die Anzahl der Paramter hängt von der Art der gewünschten Anwendung ab - üblich sind ein, zwei oder drei Parameter, je nachdem, ob man sich im $\mathbb{R}$, $\mathbb{R}^2$ oder im $\mathbb{R}^3$ befindet. Diese Einführung betrachtet nur den eindimensionalen Fall, er lässt sich aber leicht auf weitere Dimensionen erweitern.

Es gibt verschiedene Möglichkeiten, solche Pseudozufallsfunktionen zu definieren. Ist die maximale Frequenz bekannt, so kann man einfach eine Liste mit festen Zufallswerten nehmen. Die dazwischen liegenden Werte lassen sich dann interpolieren. Ist die maximale Frequenz dagegen nicht bekannt oder soll sich die Frequenz beliebig steigern lassen, benötigt man eine Funktion, die in Abhängigkeit von einem oder mehreren Parametern (im zwei- oder dreidimensionalen Fall z. B. eine räumliche Position) Werte liefert. Mehr dazu in Abschnitt 4.1.3.

Interpolation

Um aus den diskreten Funktionen eine kontinuierliche zu erzeugen, müssen die Werte zwischen den diskreten Werten interpoliert werden. Es bieten sich dafür

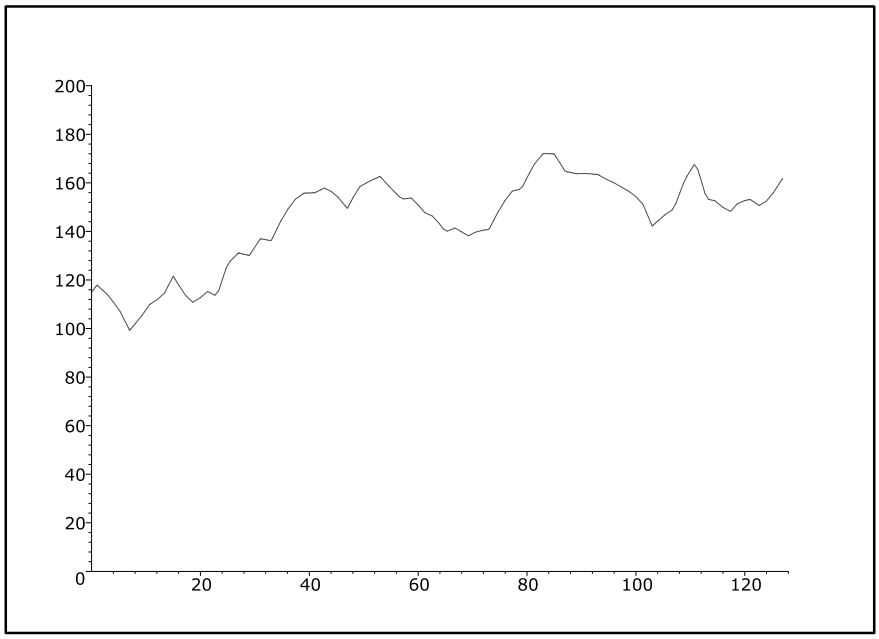

Abbildung 3.8: Summe der Rauschfunktionen (Perlin Noise)

unterschiedliche Möglichkeiten, die je nach Anforderung mehr oder weniger zufriedenstellende Ergebnisse liefern. Gewöhnlich hat eine Interpolationsfunktion drei Parameter, die beiden Werte a und b, zwischen denen interpoliert werden soll und einen Faktor x zwischen 0 und 1, welcher den Einfluss der beiden jeweiligen Werte angibt. Ist $x = 0$, liefert die Interpolationsfunktion den Anfangswert a, ist $x = 1$ liefert sie den Endwert b.

Die einfachste Möglichkeit ist eine lineare Interpolation:

$$f(x) = a \cdot (1 - x) + b \cdot x$$

Das Ergebnis ist eine stückweise definierte lineare Funktion mit einer sehr gezackten Kurve. Wählt man natürliche Strukturen als Maßstab, ist die resultierende Kurve im Allgemeinen nicht akzeptabel. Der Vorteil dieser Interpolation liegt aber in ihrer einfachen Berechnung.

Etwas aufwendiger ist eine Cosinus Interpolation. Dabei ist die Interpolationsfunktion definiert als:

$$f(x) = a \cdot \left(1 - \frac{1 - \cos(x \cdot \pi)}{2}\right) + b \cdot \left(\frac{1 - \cos(x \cdot \pi)}{2}\right)$$

Hier entsteht eine Kurve mit glatten, weichen Übergängen zwischen den einzelnen Teilstücken was im Endergebnis meist wesentlich natürlicher wirkt. Durch

die Verwendung der Cosinus Funktion ist die Berechnung aber schon deutlich aufwendiger als bei einer Linearen Interpolation. Abbildung 3.9 stellt die beiden Varianten gegenüber. Auch eine Kubische Interpolation ist möglich, doch ist ihre Berechnung extrem aufwendig und die „Verbesserung“ den Aufwand im Allgemeinen nicht Wert.

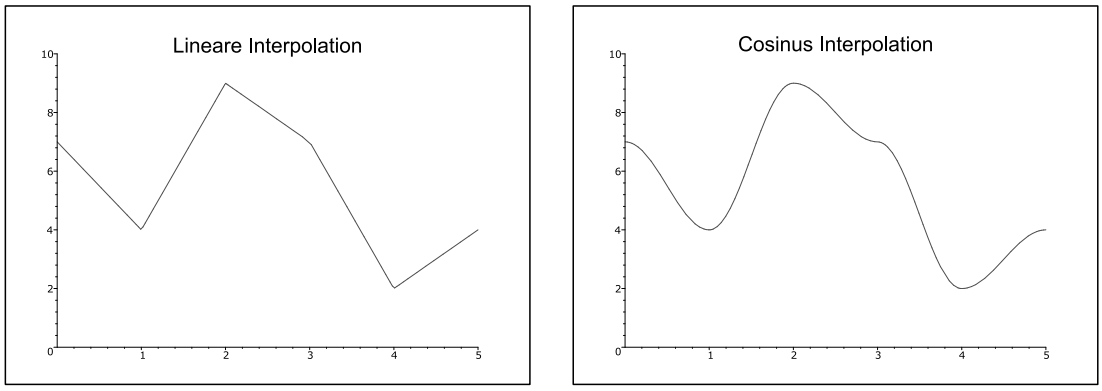

Abbildung 3.9: Lineare und Cosinus Interpolation

Persistenz

Die oben aufgeführte Veränderung der Frequenzen und Amplituden der Rauschfunktionen ist aber nicht die einzig Mögliche. Stattdessen kann man mit der Persistenz arbeiten, die wie folgt definiert[4] ist:

$$Frequenz = 2^i$$

$$Amplitude = Persistenz^i$$

Wobei i, die i. Rauschfunktion ist, welche dazu addiert wird. Das oben aufgeführte Beispiel ergibt sich nun mit einer Persistenz von 1/2. Bei einer Persistenz von 1 bleibt die Amplitude konstant, bei Werten darüber nimmt sie sogar zu. Dies führt zu extrem gezackten Kurven, die für den praktischen Einsatz wenig Nutzen bieten. Hat die Persistenz einen Wert nahe 0, sinkt die Amplitude rapide ab und nach wenigen Schritten gibt es keine nennenswerten Veränderungen mehr. Die entstehende Kurve hat einen sehr gleichmäßigen, sanften Verlauf.

[4]Dies ist nicht Benoît Mandelbrots Definition der Persistenz.

3.2.2 Midpoint Displacement

Eine andere Möglichkeit, Fraktale Landschaften zu erzeugen, ist Midpoint Displacement. Midpoint Displacement Techniken zur Erzeugung von Gelände wurden zuerst von Fournier, Fussell, und Carpenter vorgestellt[14]. Wie Perlin Noise, lässt es sich in mehreren Dimensionen nutzen. Der Einfachheit halber wird hier nur der eindimensionale Fall gezeigt. Die zweidimensionale Anwendung kann man bei Plasma bzw. beim Diamond-Square-Algorithmus sehen.

Midpoint Displacement arbeitet nach folgendem einfachen Algorithmus:

```
beginne mit einem einzelnen Liniensegment
wiederhole bis die gewünschte Auflösung erreicht ist {
  wiederhole für jedes Liniensegment {
    finde den Mittelpunkt des Liniensegments
    verschiebe ihn in Richtung y um einen zufälligen Wert
    reduziere den Wertebereich der Zufallszahlen
  }
}
```

Die folgende Abbildung veranschaulicht die Vorgehensweise, indem an einem Liniensegment drei Schritte des Algoritmus durchgeführt werden.

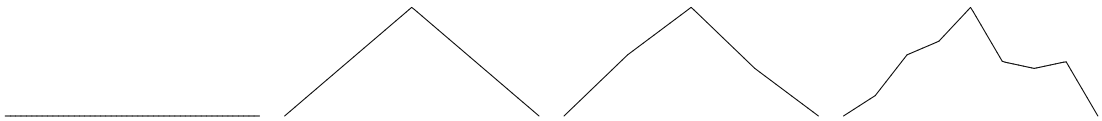

Abbildung 3.10: Midpoint Displacement in 3 Schritten

Dabei ist zu beachten, dass der Wertebereich sowohl positive, wie auch negative Zahlen enthalten kann (z. B. $[-1, 1]$). Die Verkleinerung des Wertebereichs, aus dem der Versatz des Mittelpunktes bestimmt wird, beeinflusst die Enstehung der Kurve entscheidend. Verringert man den Wertebereich in jedem Schritt gar nicht oder nur wenig, erhält man eine stark gezackte Kurve mit großen Schwankungen. Wird der Wertebereich stark verkleinert, ist die Kurve relativ glatt und weist nur geringe Schwankungen auf. Insgesamt weist die Reduzierung des Wertebereichs die gleichen Eigenschaften wie die Persistenz bei Perlin Noise auf. Eine weitere Gemeinsamkeit von Midpoint Displacement und Perlin Noise ist, die Möglich-

keit statt Linearer Interpolation (wie in Abbildung 3.10), Cosinus oder Kubische Interpolation zu verwenden.

Plasma

Plasmafraktale werden zwar im Allgemeinen eingesetzt, um Bilder bzw. Texturen (z. B. für Wolken) zu erzeugen. Sie lassen sich aber auch zur Geländemodellierung einsetzen. Das Prinzip des Midpoint Displacement wird auf den zweidimensionalen Fall übertragen. Gestartet wird hier mit einem Rechteck (idealerweise ein Quadrat). Der Midpoint Displacement Algorithmus wird wie folgt erweitert:

```
beginne mit einem einzelnen Rechteck
weise jedem Eckpunkt eine Höhe zu
wiederhole bis die gewünschte Auflösung erreicht ist {
  wiederhole für jedes Rechteck {
    wiederhole für jeder Seite des Rechtecks{
      finde den Mittelpunkt
      interpoliere seine Höhe aus der Höhe der Eckpunkte der Seite
      addiere zur Höhe einen zufälligen Wert
    }
    finde den Mittelpunkt des Rechtecks
    interpoliere seine Höhe aus der Höhe der Eckpunkte des Rechtecks
    addiere zur Höhe einen zufälligen Wert
    reduziere den Wertebereich der Zufallszahlen
  }
}
```

Die folgende Abbildung zeigt das Ergebnis von drei Schritten des Plasma-Algorithmus.

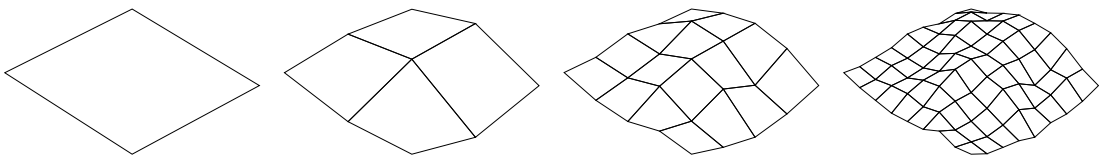

Abbildung 3.11: zweidimensionales Midpoint Displacement

Nach i Schritten wird das Rechteck in 2^{2i} Rechtecke zerlegt. Die Bestimmung der Höhe der Mitte des Rechtecks bzw. einer Seite aus der Höhe der Eckpunkte ist im Allgemeinen eine lineare Interpolation, also

$$h_M = \frac{h_A + h_B + h_C + h_D}{4} \text{ bzw. } h_M = \frac{h_A + h_B}{2}$$

falls h_A, h_B, h_C und h_D die Höhen der Eckpunkte A, B, C und D sind.

Interpretiert man Höhenwerte als Farben kann man das Ergebnis zum Beispiel als Textur verwenden.

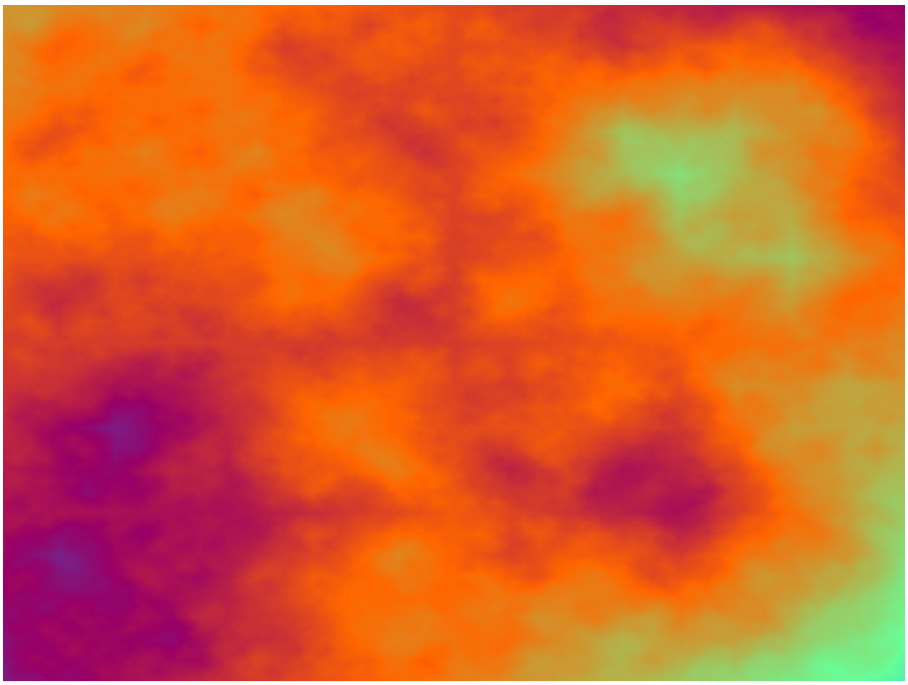

Abbildung 3.12: Plasmafraktal

Diamond-Square

Der Diamond-Square-Algorithmus nutzt Quadrate als Basis und arbeitet in zwei Schritten, dem Diamond- und dem Square-Step. Im Square-Step wird der Mittelpunkt eines Quadrates bestimmt, im Diamond-Step wird der Mittelpunkt des Diamanten ermittelt, der von den Mittelpunkten vier benachbarter Quadrate gebildet wird. Wiederholt man die Anwendung von Square- und Diamond-Schritten, wird das Quadrat suksessiv in eine Fläche von kleinen Quadraten zerlegt. Der Diamond-Square-Algorithums arbeitet also wie folgt:

Square Step:

```
finde den Mittelpunkt eines gegebenen Quadrats
interpoliere seine Höhe aus der Höhe der Eckpunkten der Seite
addiere zur Höhe einen zufälligen Wert
```

Diamond Step:

```
finde den Mittelpunkt eines gegebenen Diamanten
interpoliere seine Höheaus der Höhe der Eckpunkten der Seite
addiere zur Höhe einen zufälligen Wert
```

Diamond-Square-Algorithmus:

```
beginne mit einem einzelnen Quadrat
weise jedem Eckpunkt eine Höhe zu
wiederhole bis die gewünschte Auflösung erreicht ist {
  führe einen Square-Step für alle Quadrate durch
  führe einen Diamond-Step für alle Diamanten durch
  reduziere den Wertebereich der Zufallszahlen
}
```

Abbildung 3.13 veranschaulicht die Arbeitsweise des Diamond-Square-Algorithmus. Es werden zwei Schritte gezeigt. Dabei sind die Diamanten bzw. Quadrate, die im jeweiligen Schritt genutzt werden blau, die daraus ermittelten Punkte rot dargestellt.

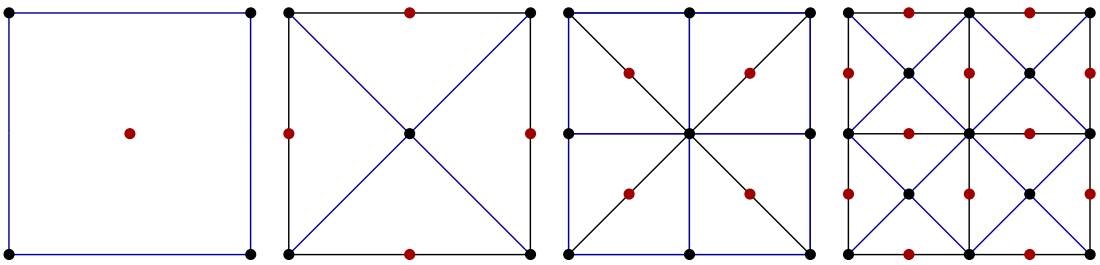

Abbildung 3.13: Diamond-Square-Schritte

Der Unterschied zu Plasma besteht nun darin, dass Plasma den Wert für die Höhe des Mittelpunktes eines Rechtecks zwar aus allen vier Eckpunkten interpoliert, die Werte für eine Seite aber nur aus zwei Eckpunkten (dem Anfangs- und

dem Endpunkt der Seite). Dies kann zu *aliasing*-ähnlichen Artefakten führen, vorallem, wenn man entlang der Gitterlinien blickt. Dieses Problem behebt der Diamond-Square-Algorithmus durch den Diamond-Step, indem nicht nur die Werte der Endpunkte einer Seite in die Interpolation mit eingehen, sondern auch die der Mittelpunkte der Quadrate, zu denen diese Seite gehört. In Abbildung 3.14 werden die Vorgehensweise bei Plasma und Diamond-Square gegenübergestellt. Der zu ermittelnde Seitenmittelpunkt ist wiederum rot, die an dessen Berechnung beteiligten Punkte bzw. Seiten sind blau dargestellt.

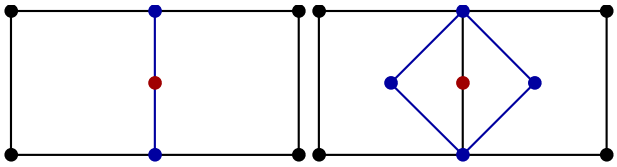

Abbildung 3.14: Seitenmittelpunktberechnung bei Plasma und Diamond-Square

Kapitel 4

Algorithmus und Implementierung

Ziel der Diplomarbeit war es, einen Algorithmus zu entwickeln, der virtuelle Planeten generiert und diese in Echtzeit auf dem Bildschirm darstellen kann. Obwohl *raytracing* qualitativ sehr gute Ergebnisse liefert und sich damit fotorealistische Bilder erzeugen lassen, scheidet es als Darstellungsoption aus.

Für die Projektive Ausgabe stehen verschiedene Grafikbibliotheken zur Verfügung, von denen OpenGL vom OpenGL Architecture Review Board und Direct3D, das Teil von Microsofts DirectX ist, die bekanntesten sind. Beide unterstützen die Hardware-Beschleunigung moderner Grafikkarten, so dass deren Fähigkeitkeiten voll ausgenutzt werden. Der Algorithmus ist von der Wahl der Bibliothek unabhängig. Für die Referenzimplementierung wurde OpenGL verwendet, da es im Gegesatz zu Direct3D für eine Vielzahl von Betriebsystemen und Plattformen zur Verfügung steht.

Mit Java3D steht zwar eine weitere plattformunabhängige Grafikbibliothek zur Verfügung, die zudem deutlich komfortabler zu programmieren ist und z. B. sogar einen Szenengraph integriert. Allerdings kann Java3D unter dem Gesichtspunkt der Performance und Geschwindigkeit nicht mit OpenGL vergleichen. Gerade unter der Bedingung der Echtzeitfähigkeit ist es deshalb keine Alternative zu Direct3D und OpenGL.

4.1 Der Algorithmus

Im Folgenden wird der entwickelte Algorithmus vorgestellt und seine Wirkungsweise beschrieben. Zuerst wird gezeigt, wie man eine Polygondarstellung für eine Kugel erstellt. Dabei werden zwei Möglichkeiten vorgestellt und begründet, warum eine dieser Möglichkeiten gewählt bzw. die andere verworfen wurde. Dann wird das Verfahren näher beschrieben, das sich in zwei Teile gliedert.

In einem LOD-Schritt wird der darzustellenden Bereich eingeschränkt, ohne ihn wäre die Echtzeitfähigkeit des Verfahrens nicht zu gewährleisten. Dabei wird untersucht, ob sich die im Abschnitt 3.1 LOD-Algorithmen für das vorgestellte Verfahren eignen und gezeigt, dass diese hier nicht eingesetzt werden können. Da aber ohne eine Form des LOD die Visualisierung des Planeten unmöglich wäre, wird anschließend eine Möglichkeit vorgestellt, wie man dennoch eine Art Level-Of-Detail einsetzen kann, um die Echtzeitfähigkeit sicherzustellen.

Gleichzeitig mit der Erzeugung der Kugel muss die eigentliche Landschaftsbildung realisiert werden. Dazu wird eine Variante des Plasma-Algorithmus eingesetzt, die an die Gegebenheiten des Basiskörpers angepasst wurde. Abschließend wird noch eine Möglichkeit beschrieben, wie man die Polarregionen eines Planeten berechnen und weiß einfärben kann, um Eis und Schnee zu simulieren.

4.1.1 Kugeldarstellung

Eine Kugel als solche lässt sich in der Computergrafik nicht exakt mit Hilfe von Polygonen darstellen. Es lassen sich aber Approximationen beliebiger Genauigkeit erzeugen. Die naheliegendste Variante ist die Darstellung mit Hilfe der Kugelkoordinaten. Dabei werden die zwei Raumwinkel φ und θ genutzt. Die Koordinaten eines jeden Punktes auf der Oberfläche der Kugel ergeben sich aus folgenden Gleichungen:

$$x = r \cdot \sin(\theta) \cdot \cos(\varphi)$$

$$y = r \cdot \sin(\theta) \cdot \sin(\varphi)$$

$$z = r \cdot \cos(\theta)$$

wobei $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ und $0 \leq \varphi \leq 2\pi$ und r der Kugelradius ist. Eine Polygondarstellung einer Kugel erhält man nun in dem θ und φ in Schritten von n bzw.

m ° innerhalb ihrer Grenzen laufen lässt. Je kleiner n und m, desto höher ist die Polygonzahl und genauer die Annäherung an eine richtige Kugel. Abbildung 4.1 zeigt eine solche Kugelapproximation durch Kugelkoordinaten.

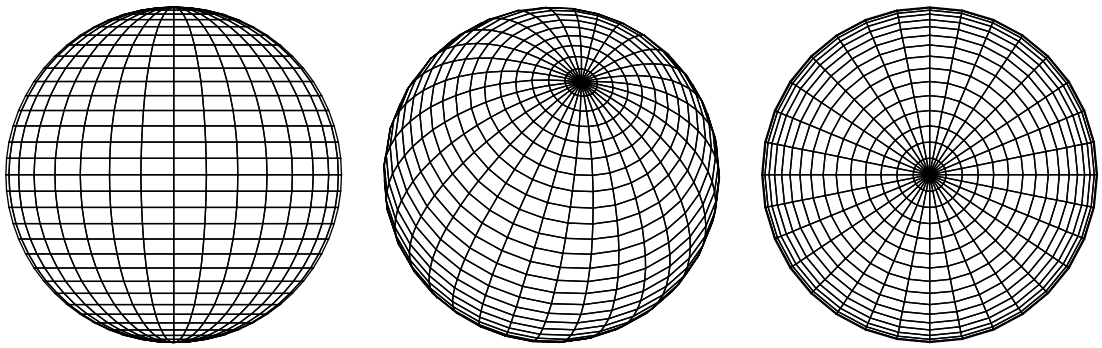

Abbildung 4.1: Kugeldarstellung aus Kugelkoordinaten

Wie man aber in der Abbildung 4.1 sieht, nimmt bei den entstandenen Oberflächenstücken die Größe zu den Polen hin ab. Folge dieser Singularität ist, dass an den Polen wesentlich mehr Polygone als z. B. am Äquator liegen. Damit scheidet diese Variante als Grundlage des zu entwickelnden Verfahrens aus, denn es würden Planeten entstehen, an deren Polen sehr feinaufgelöste, detailierte Landschaften modelliert würden. Auf dem Rest des Planeten würden dann verhältnismäßig grobe Strukturen entstehen. Eine uniforme Verteilung der Polygone, bei der alle Oberflächen der entstandenen Kugel die gleiche Größe aufweisen, wäre deshalb wünschenswert. Man kann versuchen, die an den Polen auftretende Singularität zu minimieren, indem mehrere kleine Dreiecke/Vierecke zu einem zusammenfasst werden, so dass ein Dreieck/Viereck entsteht, dessen Größe mit der der Restlichen vergleichbar ist.

Eine andere Möglichkeit ist, die Kugel durch weniger komplexe geometrische Körper zu approximieren. Die einfachste Approximation[1] stellt der Tetraeder dar. Ein sehr kleiner oder sehr weit entfernter Tetraeder sieht für einen Betrachter wie eine Kugel aus. Man kann diese grobe Näherung verbessern, indem man gleichmäßig weitere Eckpunkte hinzufügt. Gleichmäßig heißt hier, dass die symmetrische Form des Körpers erhalten bleibt. Dies wird erreicht, in dem jedes Dreieck ABC des Tetraeders durch folgende Dreiecke ersetzt wird[2]:

[1] Einfach heißt hier, gemessen an der Anzahl der Polygone.

[2] Unter der Vorraussetzung, dass sich der Mittelpunkt des Tetraeders im Koordinatenur-

Seien $\overrightarrow{a}$, $\overrightarrow{b}$ und $\overrightarrow{c}$ die Vektoren vom Koordinatenursprung zu den Eckpunkten des Dreiecks, dann sind $\overrightarrow{d} = \overrightarrow{a} + \overrightarrow{b}$, $\overrightarrow{e} = \overrightarrow{a} + \overrightarrow{c}$ und $\overrightarrow{f} = \overrightarrow{b} + \overrightarrow{c}$ neue Vektoren mit den dazugehörigen Eckpunkten. Ihre Entfernung zum Koordinatenursprung entspricht aber noch nicht dem Radius der gewünschten Kugel. Das lässt sich leicht erreichen, wenn man den Vektor zuerst normalisiert und mit dem gewünschten Radius skaliert. Das Dreieck ABC wird ersetzt durch die Dreiecke ADE, EDF, DBF und EFC.

Wendet man dieses Verfahren auf alle Seiten des Tetraeders an, erhält man als Ergebnis einen Körper mit 16 Begrenzungsflächen, der schon deutlich eher einer Kugel ähnelt. Wiederholt man das Verfahren für jede dieser 16 Seiten, erhält man einen Körper mit 64 Flächen. Mit jeder weiteren Iteration vervierfacht sich die Anzahl der Dreiecke und man nähert sich immer weiter an eine Kugel an. Dieses, *Surface-Refinement* [6] genannte Verfahren, lässt sich auch sehr einfach rekursiv implementieren.

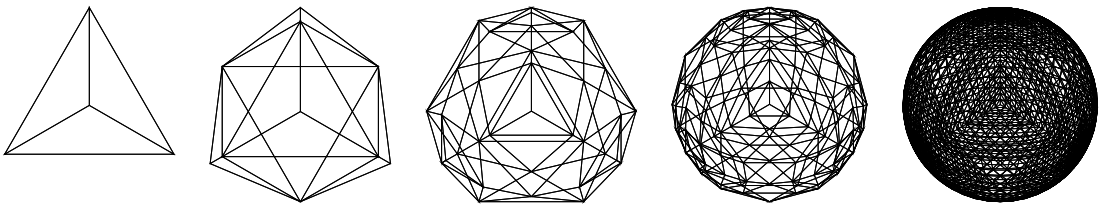

Abbildung 4.2: Surface Refinment am Tetraeder

Abbildung 4.2 zeigt wie mit *Surface-Refinement* ein Tetraeder in 5 Schritten in eine Kugel verwandelt wird. Dieses *Surface-Refinement* funktioniert aber nicht nur beim Tetraeder, sondern auch beim Oktaeder, Ikosaeder und sogar beim Hexaeder[3]. Allerdings muß beim Hexaeder die Vorgehensweise leicht verändert werden, da seine Seitenflächen aus Quadraten und nicht aus Dreiecken bestehen.

Zwar führt auch dieses Verfahren nicht zu einer uniformen Verteilung der Polygone bzw. Vertizes, wie Abschnitt 5.1 zeigt, dennoch sind die resultierenden Kugeln deutlich besser geeignet als jene, die durch Kugelkoordinaten gewonnen werden können.

Aus diesem Grunde stand zu Beginn der Diplomarbeit die Idee, einen Tetraeder als Grundlage für das zu entwickelnde Verfahren zu wählen. Im Zuge

sprung befindet.

[3]dem Würfel

der Entwicklung wurden aber immer mehr Schwächen und Unzulänglichkeiten[4] festgestellt, die durch die geometrischen Eigenschaften dieses Körpers begründet sind. Dies führte über einen Zwischenschritt mit einem Oktaeder zu der endgültigen Entscheidung, einen Ikosaeder als Basis zu wählen. Dieser wird schrittweise mit dem *Surface-Refinement*-Verfahren in eine Kugel umgewandelt. Gleichzeitig werden für jeden neuentstandenen Punkt Höhenwerte berechnet, die die Oberfläche des Planeten formen. In Abschnitt 3.2 wurden einige Möglichkeiten dazu vorgestellt.

4.1.2 Der LOD-Teil

Bevor jedoch mit der Geländemodellierung überhaupt begonnen werden kann, wird in einem ersten Schritt der darzustellende Bereich eingeschränkt und der Detailgrad der Darstellung bestimmt. Beide hängen unmittelbar vom Standpunkt des Betrachters bzw. der virtuellen Kamera ab. Ist der Betrachter der Planetenoberfläche sehr nahe, muss der sichtbare Bereich sehr fein aufgelöst werden, um eine angemessene Darstellungsqualität zu erreichen. Gleichzeitig kann aber auch ein Großteil der Planetenoberfläche von der Darstellung ausgeschlossen werden. Man kann nämlich davon ausgehen, dass ein Betrachter, der sich direkt auf der Planetenoberfläche befindet, nur seine unmittelbare Umgebung, nicht aber andere Kontinente oder die gegenüberliegende Seite des Planeten sehen kann.

Ist der Betrachter dagegen sehr weit von der Planetenoberfläche entfernt, wird der Planet als Ganzes sichtbar und muß dem entsprechend auch komplett berechnet und dargestellt werden. Allerdings genügt nun eine grob aufgelöste Darstellung, denn kleine Details der Planetenoberfläche sind aus großer Entfernung nur schwer oder gar nicht zu erkennen. Im Idealfall entspricht die zu berechnede Auflösung der Auflösung des Ausgabemediums (z.B. Röhren- oder LCD-Monitore), so dass ein Dreieck einem Pixel entspricht. Bei Supersampling liegt sie noch darüber. Dies ist mit dem heutigen Stand der Technik nicht erreichbar. Deshalb ist es notwendig, die Zahl der Polygone auf ein mit der heutigen Technik handhabbares Maß zu reduzieren und dabei trotzdem größtmögliche Qualität zu erreichen. Dies geschieht mit Hilfe von LOD-Verfahren, wie sie in Abschnitt 3.1 vorgestellt wurden.

[4] siehe dazu vor allem Abschitt 5.1

Lindstrom, Röttger, ROAM?

In Abschnitt 3.1 wurden einige sehr vielversprechende Algorithmen vorgestellt, die das Visualisieren von großen Landschaften überhaupt erst möglich machen. Leider lassen sie sich nicht ohne Weiteres mit dem *Surface-Refinement* kombinieren.

Lindstrom/Röttger Da Lindstrom und Röttger sehr ähnlich sind, Röttger aber gegenüber Lindstrom durch seine *top-down*-Strategie und sein Geomorphing deutlich besser ist, wird hier nur Röttger näher betrachtet. Lindstrom und Röttger setzen auf quadratische Blöcke, die sie mit Hilfe von *quad trees* rekursiv verfeinern, während *Surface-Refinement* gleichseitige Dreiecke als Basis hat. Immerhin ist beiden gemeinsam, dass jede Fläche rekursiv in vier Teilflächen unterteilt wird. Auch die beiden Entscheidungskriterien von Röttger (siehe dazu noch einmal Abschnitt 3.1.3) lassen sich noch auf diese Situation anpassen. Zwar ist der Mittelpunkt bei einem gleichseitigen Dreieck etwas aufwendiger zu bestimmen, als bei einem Quadrat, dafür müssen zur Bestimmung der Oberflächenbeschaffenheit nur drei statt sechs Berechnungen durchgeführt werden.

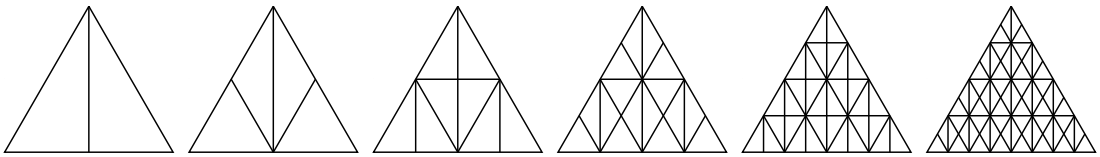

Abbildung 4.3: ROAM-Triangulierung auf gleichseitigen Dreiecken

ROAM ROAM setzt bei der Verfeinerung der Auflösung des Geländes wie *Surface-Refinement* auf Dreiecke. Allerdings nutzt ROAM dabei rechtwinklige, gleichschenklige Dreiecke, während *Surface-Refinement* gleichseitige Dreiecke setzt. Nun sind zwar gleichseitig Dreiecke auch gleichschenklig, allerdings gilt bei gleichseitigen Dreiecken, dass die Größe aller Innenwinkel $\alpha = 60\ ^\circ$ ist. Es gibt also keinen rechten Winkel. Die Zerlegungsstrategie von ROAM kann also hier nicht angewendet werden, da nicht alle nach der Unterteilung entstehenden Dreiecke gleichseitig oder gleichschenklig sind, wie Abbildung 4.3 zeigt. Interessant ist dabei, dass nach einem weiteren Unterteilungsschritt insgesammt die selben Punkte

ausgewählt werden wie beim *Surface-Refinement* und im nächsten Schritt sogar eine ähnliche Triangulierung entsteht. Lediglich die vertikalen Kanten fehlen beim *Surface-Refinement*. Abbildung 4.3 zeigt die Anwendung der ROAM-Triangulierung auf ein gleichseitiges Dreieck.

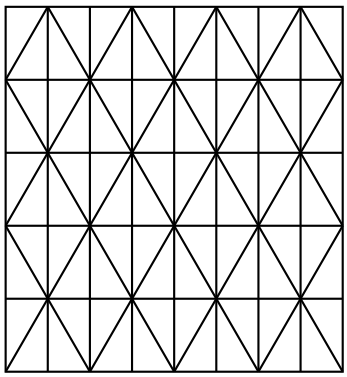

Abbildung 4.4: Patch im ROAM-Triangulierung

Dennoch ist dieses Ergebnis unbefriedigend. Die entstandene Triangulierung ist zwar regelmäßig, aber nicht gleichmäßig, da die horizontalen Abstände zwischen den Punkten deutlich geringer sind als die vertikalen. Abbildung 4.4 verdeutlicht dies noch einmal an einem Patch, der durch ROAM-Refinement entstanden ist. Dieser Umstand lässt sich nur beheben, wenn die vertikalen Kanten nachträglich entfernt werden. Dann erhält man den gleichen Patch wie bei *Surface-Refinement*.

T*-Vertizes und *cracks Es gibt aber noch ein weiteres Problem, das den Einsatz der obengenannten Verfahren verhindert. An benachbarten Blöcken mit unterschiedlicher Detailstufe können Risse auftreten, da sie unterschiedlich trianguliert sind. Abbildung 4.5 zeigt einen solchen Block. Die im linken Teil blau markieren Vertizes werden von nur 3 Kanten gebildet. Diese bilden ein T und die Vertizes werden daher T-Vertizes genannt. Alle anderen Vertizes werden immer von 4 Kanten gebildet und bilden ein Kreuz. And diesen T-Vertizes kann es dann aber beim *rendering*, wie Abbildung 4.5 rechts zu sehen, zu Diskontinuitäten kommen, die der Betrachter dann als Risse in der Landschaft wahrnimmt.

Alle *LOD*-Algorithmen haben mit diesem Problem zu kämpfen und haben deshalb entsprechende Techniken entwickelt. Bei ROAM werden die gleichschenkligen Dreieck aufgeteilt, indem eine neue Kante zwischen der Mitte Basis und dem

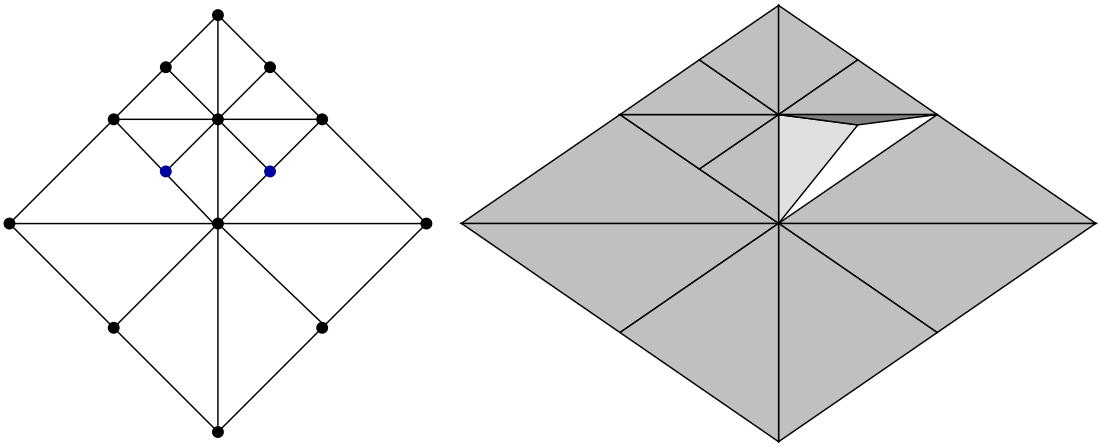

Abbildung 4.5: T-Vertizes und *Cracks*

gegenüberliegenden Vertex eingefügt wird. Um *cracks* zuvermeiden, wird deshalb nicht nur dieses Dreieck geteilt, sondern auch das der Basis gegenüberliegende. Dies kann unter Umständen weitere Teilungsoperationen notwendig machen, wie Abbildung 4.6 deutlich macht. Hier soll das blaue Dreieck aufgespalten werden. Um dies zu erreichen muss jedoch zuvor das rote Dreieck unterteilt werden, was jedoch erst möglich ist, wenn das grüne Dreieck aufgeteilt wurde. Der ganz Prozess endet erst, wenn die beiden hellblauen Dreiecke ebenfalls aufgeteilt wurden. Im schlimmsten Fall zieht also die Aufteilung eines Dreiecks die Teilung von 4 weiteren Dreiecken nach sich. Durch dieses Vorgehen wird verhindert, dass T-Vertizes entstehen.

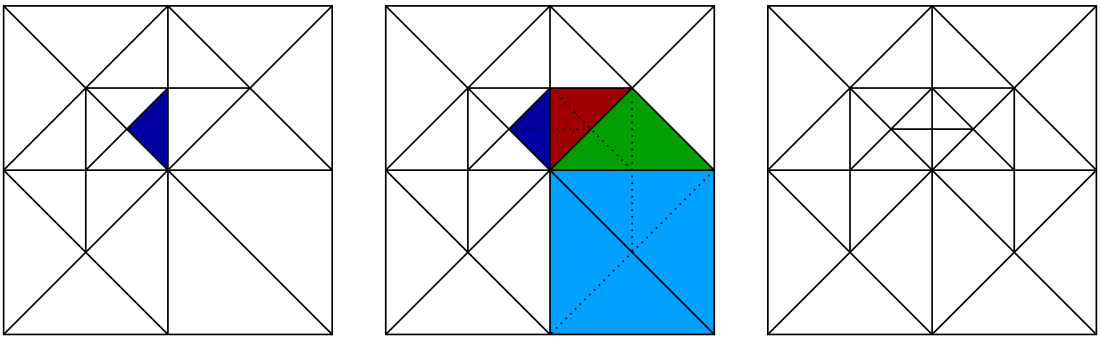

Abbildung 4.6: forced-splitting bei ROAM

Röttger wählt einen anderen Ansatz. Ist im *quad tree* ein Blatt erreicht, wird ganzer oder partieller *triangle fan* gezeichnet (siehe dazu Abbildung 4.7). Dazu wird in jedem Block vom Mittelpunkt aus eine Kante zu den vier Eckpunkten

und den Mittelpunkten der vier Seiten des Blocks eingefügt. An dieser Stelle wird geprüft, ob die benachbarten Blöcke die gleiche Detailstufe haben. Ist das nicht der Fall, wird das Vertex am Mittelpunkt der gemeinsamen Seite und damit auch die Kante weggelassen. Auch diese Methode verhindert das Auftreten von T-Vertizes und damit *cracks*.

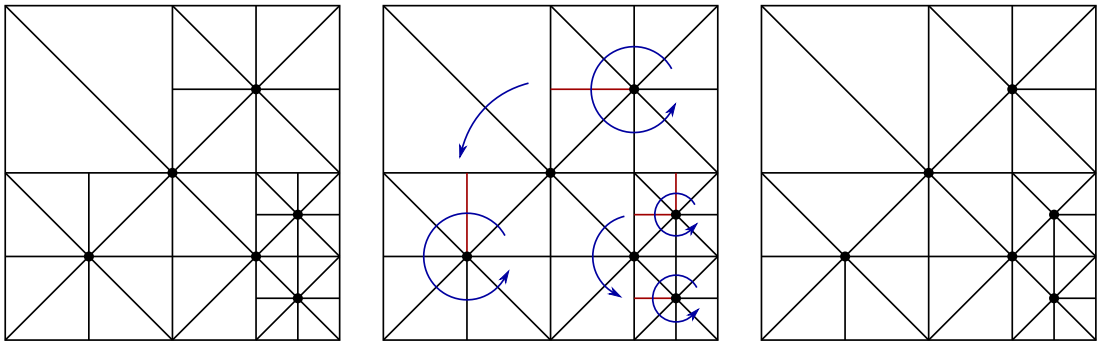

Abbildung 4.7: Entfernung der T-Vertizes bei Röttger

Beide Varianten basieren aber auf Eigenschaften, die sich nicht auf ein mit *Surface-Refinement* erzeugtes Dreiecksnetz übertragen lassen. Ursache ist, dass beim Einfügen eines neuen Vertex in der Mitte einer Kante gleich zwei neue Kanten entstehen und nicht nur eine. Es handelt sich also nicht um T- sondern um K-Vertizes. Das Einfügen eines Vertex beim *Refinement* eines Dreiecks erfordert zwingend das Einfügen der beiden anderen. Es ist also nicht wie bei quadratischen Blöcken möglich, nur einen Teil des Blocks weiter zu verfeinern, sondern ein Dreieck muß immer als Ganzes und vollständig unterteilt werden. Die Ansätze von ROAM und Röttger funktionieren hier also nicht.

Gibt es eine andere Möglichkeit *cracks* zu verhindern? Es bestünde noch die Möglichkeit, die Risse nachträglich zu stopfen, indem man an dieser Stelle ein Füllpolygon einfügt, welches den aufgetretenen Riss schließt. Da es sich dabei aber immer um senkrechte Polygone handelt, fallen diese in der Landschaft fast genauso störend auf, wie die Risse selbst. Besser wäre es an dieser Stelle, den Vertex, der den *crack* verursacht, nicht an seiner eigentlichen Position zu rendern, sondern die Höhe auf 0 zu lassen. *cracks* würden somit nicht entstehen und es ließe sich hier sogar ein Geomorphing einsetzen, um die Vertizes langsam anzuheben und das *popping*-Problem zu verringern. Diese Methode setzt allerdings vorraus, dass man K-Vertizes effektiv ermitteln kann. Dazu sind Nachbarschaftstests not-

wendig, die im Allgemeinen sehr aufwendig sind. Mit einer Baumstruktur lässt sich dieses Problem vereinfachen, allerdings auch nur dann, wenn man sicherstellt, dass sich zwei benachbarte Dreiecke nur um eine Auflösungsstufe unterscheiden, wie Röttger zeigt [12].

Da für dieses Problem keine befriedigende Lösung gefunden werden konnte, werden die Dreiecke in der Referenzimplementierung vollständig rekursiv aufgeteilt. Leider steigt damit die Zahl der Polygone und damit auch die Laufzeit exponentiell. Damit aber der Planet auch bei hohen Auflösungen noch in Echtzeit dargestellt werden kann, wird nur die unmittelbare Umgebung des Betrachters berechnet.

Erste Ansätze

Nachdem also deutlich wurde, dass die bisher bekannten LOD-Techniken nicht zusammen mit dem in Abschnitt 4.1.1 vorgestellten *Surface-Refinement* funktionieren, musste ein anderer Ansatz gefunden werden, die Polygonzahl bei der Darstellung zu begrenzen. Vor allem, da die Polygonzahl sich in jedem *Refinement*-Schritt vervierfacht und somit exponientiell wächst. Es liegt auf der Hand, dass in einer Diplomarbeit nicht ein mit ROAM oder Röttger vergleichbares Verfahren als Nebenprodukt entwickelt werden kann. Dennoch wurde eine einfache Möglichkeit gefunden, den darzustellenden Bereich des Planeten einzuschränken und so die Polygonzahl in Abhängigkeit vom Betrachterstandpunkt zu reduzieren.

Schon bei ersten Überlegungen zu dieser Problematik kam die Idee auf, bei den ersten Rekursionschritten des *Surface-Refinement* bestimmte Seiten wegzulassen und somit die zu berechnende Fläche zu beschränken. Dabei war angedacht, in jedem Schritt die vorhandenen Dreiecke gemäß der im Abschnitt 4.1.1 beschriebenen Art und Weise zu unterteilen und dann festzustellen, welche der vorhandenen Seiten sich direkt zwischen Betrachter und Zentrum des Planeten befindet. Nur diese Seite sollte dann anschließend weiter unterteilt werden, die restlichen Seiten wären von der weiteren Berechnung ausgeschlossen. Dieser Vorgang wird nun eine bestimmte Anzahl von Schritten wiederholt, anschließend wird mit rekursivem *Surface-Refinement* fortgefahren.

Diese Vorgehensweise führt aber schnell zu Problemen, denn der rekursive Teil des Algorithmus startet nun mit einem Dreieck. Das Ergebnis ist eine sphärische Projektion dieses Dreiecks. Nähert sich nun der Betrachter dem Rand dieses Aus-

gangsdreiecks, ändert sich der berechnete Ausschnitt nicht. Der vorgeschlagene Algorithmus wird immer wieder das gleiche Dreieck als Ausgangpunkt wählen, bis der Betrachter es verlässt und ein benachbartes Dreieck betritt. Das ist natürlich viel zu spät, da der Betrachter diesen Teil des Planeten bereits vorher sehen würde und nicht erst, wenn er ihn betritt. Ein weiterer Nachteil dieser Variante ist, dass, selbst wenn sich der Betrachter im Mittelpunkt des Ausgangsdreiecks befindet, die Seitenmittelpunkte des Dreiecks relativ nah, die Eckpunkte dagegen relativ fern sind[5]. Idealerweise sollten aber alle Randpunkte des darzustellenden Areals gleich weit vom Betrachter entfernt sein.

Verbesserung

Die dargelegten Schwächen führten zu einer verbesserten Strategie zur Auswahl der wegzulassenden Seiten. Wie bereits gezeigt, genügt es nicht, nur dasjenige Dreieck auszuwählen, auf dem der Betrachter „steht“. Auch die benachbarten Seiten müssen in die Berechnung mit einbezogen werden. Dies führte zu dem neuen Ansatz, nicht mehr das Dreieck zu wählen, welches sich direkt zwischen dem Betrachter und Planetenmittelpunkt befindet, sondern den Eckpunkt, der dem Betrachter am nächsten ist und mit ihm alle Dreiecke auszuwählen, die diesen Punkt als Eckpunkt haben. Im Allgemeinen Fall ist dies nun ein geschlossenes Dreiecksnetz[6] bestehend aus 6 Dreiecken, die sich konzentrisch um den zum Betrachter nächstgelegenen Punkt befinden. Damit sind die oben angeführten Probleme gelöst. Allerdings muss nun ein Weg gefunden werden, den Abstand von zwei Punkten im dreidimensionnalen Raum zu bestimmen und das möglichst effizient. Aufwendige Berechnungen, wie z. B. Schnittpunktberechnungen, würden die Echtzeitfähigkeit des Verfahrens gefährden.

Es gibt aber zum Glück eine sehr einfache Methode, um zu einem gegebenen Punkt P aus einer Menge von anderen Punkten, denjenigen zu finden, der den geringsten Abstand hat. Die Koordinaten eines Punktes im $\mathbb{R}^2$ oder $\mathbb{R}^3$ kann man auch als einen Vektor auffassen. Für zwei Punkte A und B mit dazugehörigen Vektoren $\vec{a}$ und $\vec{b}$ entspricht dann der Abstand zwischen ihnen der Länge eines Vektors von Punkt A nach Punkt B. Dieser Vektor kann aber leicht als die

[5] Die Entfernung der Seitenmittelpunkte entspricht dem Radius des Inkreises, die der Eckpunkte entspricht dem Radius des Umkreises des Ausgangsdreiecks.

[6] genauer gesagt ein *triangle fan*

Differenz der Vektoren $\vec{a}$ und $\vec{b}$ gebildet werden. Abbildung 4.8 erläutert dies für den zweidimensionalen Fall genauer. Sei also $\vec{c} = \vec{a} - \vec{b}$ dann ist $|\vec{c}| = \sqrt{c_x^2 + c_y^2 + c_z^2}$ die Länge des Vektors und damit der Abstand zwischen den Punkten A und B, wobei c_x, c_y und c_z jeweils die x-, y- und z-Komponente des Vektors $\vec{c}$ sind. Damit lassen sich mit vergleichsweise einfachen Operationen Abstände zwischen beliebigen Punkten im zwei- und dreidimensionalen Raum bestimmen.

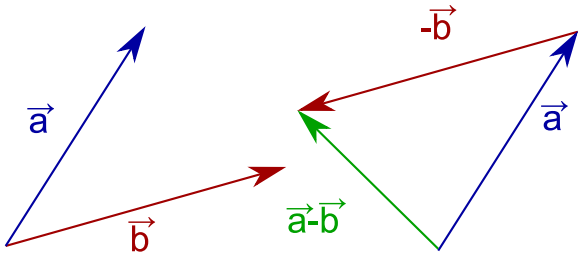

Abbildung 4.8: Vektorsubtraktion zur Entfernungsbestimmung

Der LOD-Teil sieht im Pseudo-Code also folgendermaßen aus. Begonnen wird dabei mit einem gegebenen Dreiecksnetz (z.B. in Form einer Liste) und der Position des Betrachters bzw. der virtuellen Kamera.

```
bilde den Vektor zur Betrachterposition
wiederhole für eine gegebene Anzahl von Schritten {
  setze die bisherige Minimalentfernung auf unendlich
  setze nächstgelegen Vektor auf 0
  ermittle alle Punkte bzw. Vektoren aus dem Dreiecksnetz
  wiederhole für jeden Vektor {
    subtrahiere ihn vom Vektor zur Betrachterposition
    bilde davon den Betrag
    ist der Betrag kleiner als die aktuelle Minimalentfernung {
      merke diesen Vektor als neuen nächstgelegenen Vektor
      setze neue Minimalentfernung auf den Betrag des Vektors
    }
  }
  bestimme den zum nächstgelegenen Vektor gehörenden Punkt
  bestimme alle Dreiecke, diesen Punkt als Eckpunkt haben
  setze diese Dreiecke als neues Dreiecksnetz
}
```

Das Resultat dieses LOD-Teils ist nun ein Dreiecksnetz mit besonderer Form. Es besteht im Allgemeinen[7] aus sechs Dreiecken, die alle einen gemeinsamen Punkt haben. Dieser bildet das Zentrum des Dreiecksnetzes.

Die Entfernung d des Betrachters zum Zentrum des rekursiv berechneten Areals beträgt maximal $\frac{a}{2}\sqrt{3}$, wobei a die Seitenlänge des Dreiecks ist, in dem sich der Betrachter am Beginn des Rekursionsstarts befindet. Diese Seitenlänge wird in jedem Schritt in etwa halbiert (siehe dazu Abschnitt 5.1)

4.1.3 Landschaftsmodellierung

In Abschnitt 3.2 wurden zwei grundsätzliche Verfahren vorgestellt, mit denen sich mit fraktalen Algorithmen Landschaften modellieren lassen. Da der *Fractal-Plasma*-Algorithmus, wie *Surface-Refinement*, auf einer rekursiven Verfeinerung der Auf-lösung beruht, bietet er sich zur Terrainerzeugung gerade zu an, beide lassen sich gut miteinander kombinieren. Dazu muss lediglich bei jedem *Refinement*-Schritt für jeden neu erzeugten Punkt zusätzlich ein Höhenwert berechnet werden. Im dieses zu erreichen, wird wie bei *Plasma* die Höhe des neuen Punktes einer Seite linear aus der Höhe des Start- und des Endpunktes interpoliert und dann noch um einen zufälligen Wert verschoben.

Ein *Refinement*-Schritt kombiniert mit Landschaftserzeugung sieht dann wie folgt aus:

```
wiederhole für jedes Dreieck {
  wiederhole für jede Seite des Dreiecks {
    finde den Mittelpunkt der Seite
    skaliere ihn mit dem Radius der Kugel
    interpoliere die Höhe aus der Höhe des Start- und des Endpunktes
    verschiebe die interpolierte Höhe um einen zufälligen Betrag
  }
}
reduziere den Wertebereich der Zufallszahlen
```

[7] Sollte in jedem Schritt des LOD-Teils immer der gleiche Punkt ausgewählt werden, dann handelt es sich um einen Eckpunkt des Ausgangskörpers und die Zahl der Dreiecke entspricht derjenigen, die jeder Eckpunkt des Ausgangskörper hat - drei beim Tetraeder, vier beim Oktaeder und fünf beim Ikosaeder.

Werden die Höhenwerte tatsächlich rein zufällig berechnet, ergeben sich schnell Probleme. Durch das vorgestellte Level-of-Detail-Verfahren wird dafür gesorgt, dass niemals der gesamte Planet in voller Auflösung zu sehen ist. Wird ein Teil des Planeten im Zuge der Darstellung erneut berechnet, wird auch sein Höhenprofil erneut berechnet. Da die Höhenwerte aber zufällig ermittelt werden, ergibt sich ein anderes Höhenprofil als bei der vorhergehenden Berechnung. Der Betrachter bekommt nun eine völlig andere Landschaft zu sehen.

Eine Möglichkeit, dieses Problem zu umgehen, wäre den für jeden Punkt zufällig berechneten Wert zu speichern und bei einer erneuten Darstellung zu laden. Allerdings würden dann bei hohen Auflösungen enorme Datenmengen anfallen, die entsprechend aufwendig zu verwalten wären. Eine bessere Möglichkeit ist, keine echten Zufallszahlen zu wählen, sondern Zahlen, die lediglich zufällig erscheinen, sich aber deterministisch, in Abhängigkeit verschiedener Parameter, berechnen lassen.

Da für jeden Punkt immer der gleiche Höhenwert berechnet werden soll, die Höhenwerte für verschiedene Punkte verschieden sein und möglichst zufällig aussehen sollen, ist es naheliegend, die drei Raumkoordinaten eines jeden Punktes als Parameter in die Berechnung mit einzubeziehen. Wählt man allerdings nur diese, so entsteht bei gleichen Startkoordinaten auch immer der gleiche Planet. Um dies zu beheben, könnte man die Koordinaten des Ausgangskörpers (z. B. des Ikosaeders) vor Beginn des *Surface-Refinement* einer zufälligen Rotation um eine beliebige Achse durch den Koordinatenursprung unterziehen oder den Radius der Kugel verändern. Es ist aber einfacher, eine Funktion zu wählen, die neben den drei Punktkoordinaten noch einen vierten Parameter nutzt, der dann vor der Erzeugung des Planeten rein zufällig gewählt werden kann.

Es gibt verschiedene Möglichkeiten für solche Funktionen. So ließe sich die von Ken Perlin in [3] vorgestellte und in [15] verbesserte Noise-Funktion auf den vierdimensionalen Fall erweitern. Die dabei durchgeführten Berechnungen sind aber vergleichsweise aufwendig. Deshalb bedienen sich viele andere Verfahren (z. B. Winzen in [15] und Elias in [13]) einer Kombination von Addition, Mutliplikation sowie Shift- und Modulo-Operationen der Parameter und großer Primzahlen. Die Parameter liegen dabei als ganze Zahlen vor.

Die Referenzimplementierung verwendet eine leicht veränderte Version der von Hugo Elias eingesetzten *noise*-Funktion. Da diese nur für den zweidiemensio-

nalen Fall ausgelegt ist, wurde sie auf vier Parameter erweitert. Die verwendete Funktion ist wie folgt definiert:

```
function RandomContext(integer seed, integer x, integer y, integer z)
    n = seed + x * 57 + y * 127 + z * 307;
    n = (n<<13) ^ n;
    r = ( (n * (n * n * 15731 + 789221) + 1376312589) & 7fffffff;
    return ( 1.0 - r) / 1073741824.0);
end function
```

Dabei ist der Parameter `seed` ein beliebiger Startwert. Dieser ist für jeden Punkt gleich und bestimmt das genaue Aussehen des Planeten. Die Variablen `x`, `y` und `z` entsprechen den Koordinaten des Punktes.

Das Ergebnis dieser `RandomContext`-Funktion ist ein rein zufällig erscheinender Wert zwischen 0 und 1. Dieser wird im i. *Refinement*-Schritt zusätzlich mit 2^{-i} multipliziert, um, wie in in Abschnitt 3.2.2 gezeigt, den Wertebereich der Zufallszahlen in jedem Schritt zu verringern (im konkreten Fall zu halbieren).

4.1.4 Polkappen

Obwohl es schon eine Reihe von Programmen gibt, die Fraktale Landschaften erzeugen, beschränken sich doch fast alle mehr oder weniger auf Morphologie des Geländes. Das mag für kleine Areale durchaus ausreichend sein. Will man aber ganze virtuelle Planeten erschaffen, spielen Vegetations- und damit Klimazonen eine wichtige Rolle. Es liegt auf der Hand, warum dieses Thema bisher kaum beachtet wurde. Das Klima eines Planeten ist ein chaotischer Prozess und die Klima- und damit auch Vegetationszonen hängen von einer Vielzahl von Faktoren ab, von denen Temperatur und Feuchtigkeit sicher nur die wichtigsten sind.

Während man für die Temperatur noch eine grobe Abschätzung machen kann, ist es für die Feuchtigkeit schwierig, einfache mathematisch Modelle aufzustellen. Neben direkten morphologischen Eigenschaften spielen viele sekundäre Faktoren, wie z. B. Meeresströmungen, eine wichtige Rolle. Auch diese Diplomarbeit kann diese Fragen nur am Rande behandeln. Es wird eine Möglichkeit gezeigt, wie man zumindest schneebedeckte Polkappen und Berge darstellen kann.

Die Temperatur eines Ortes ist ist vorallem geprägt von seiner geografischen Breite und seiner Höhe. Ersteres lässt sich noch relativ leicht aus dem Winkel der

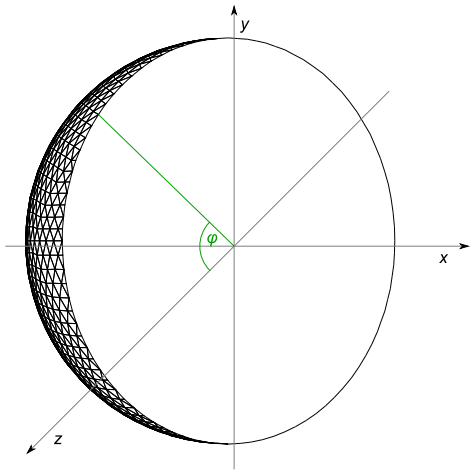

Abbildung 4.9: Breitenwinkel an einer Kugel

Sonneneinstrahlung erklären. Beim Zusammenhang zwischen Höhe und Temperatur ist nicht ganz so einfach. Immerhin lässt sich für die Troposphäre[8] feststellen, dass die Temperatur linear mit der Höhe abnimmt. Damit lassen sich aus der geografischen Breite und der Höhe eines Punktes auf der Planetenoberfläche zumindest eine grobe Abschätzung für seine Temperatur gewinnen. Ken Musgrave nutzte dies schon in seiner Dissertation [2].

Nun ist es also notwendig, die geografische Breite oder ein ähnliches Maß zu finden. Da die geografische Breite aber ein astronomisches Maß ist[9], ist sie für virtuelle Planeten nicht geeignet. Eine andere Möglichkeit ist, den Winkel eines Punktes zum Äquator bzw. zum Pol zu bestimmen, wie Abbildung 4.9 verdeutlicht. Er ist mit dem Winkel φ aus der Polarkoordinatendarstellung eines Kreises vergleichbar und wird auch wie dieser berechnet.

$$\varphi = \begin{cases} \arcsin \frac{y}{r} & \text{für } y \geq 0 \\ -\arcsin \frac{y}{r} & \text{für } y < 0 \end{cases}$$

Wobei y die y-Koordinate des Punktes ist und r der Radius der Kugel. Auf diese Art und Weise läßt sich nun leicht ein Breitenmaß für alle Punkte des Planeten berechnen. Damit kann man eine Art Klimazonenmodell einführen, indem man nun für jede Klimazone den Bereich über seine Grenzen in Form von zwei Winkeln definiert. Die Polarzone wäre dann z. B der Bereich von $90\,^\circ$ bis $66,5\,^\circ$.

[8]Die Troposphäre ist der Teil der Erdatmosphäre, in dem die Wetterbildung geschieht.

[9]Sie wird aus dem Sonnhöchststand oder aus den Sternen ermittelt.

Für jeden Punkt lässt sich nun seinen Breitenwinkel φ bestimmen und feststellen, in welcher Zone er liegt. Hierzu ist es aber immerhin noch notwendig, für jeden Punkt die trigonometrische Arcussinus-Funktion durchzuführen. Dies lässt sich vermeiden, wenn man die Grenzen nicht als Winkel definiert, sondern über den Sinus des Winkels mutlipliziert mit dem Radius r, also aus der Umkehrung der oben genannten Formel. Damit sind die Grenzen des Polarkreises einer Einheitskugel 1 und $\approx 0,917$. Das Bestimmen der Klimazone reduziert sich also für jeden Punkt auf den Vergleich der y-Koordinate mit diesen festgelegten Grenzen.

In der Referenzimplementierung werden nur die Polkappen eingezeichnet. Prinzipiell kann man mit dieser Methode aber beliebig viele „Klimazonen“ definieren. Die auf diese Weise erzeugten Zonen wirken allerdings wenig natürlich, da sie allein den Breitengrad als Maß haben und der Übergang zwischen zwei Zonen immer Parallel zum Äquator verläuft. Hier kann man zweidimensionales Midpoint-Displacement einsetzen, um die Grenze in einem Übergangsbereich fraktal zu verschieben und damit ein etwas realistischeres Aussehen zu verleihen.

4.2 Die Implementierung

Hier folgen nun einige Informationen zur Referenzimplempentierung des oben vorgestellten Algorithmus. Der eigentliche Algorithmus ist in der Klasse `Planet` implementiert. Diese enthält weder Windows- noch OpenGL-spezifischen Code und sollte sich somit leicht auf andere Plattformen, Betriebssysteme und Grafikbibliotheken übertragen lassen. Im folgenden werden die wichtigsten Methoden dieser Klasse erläutert. Der Quelltext dieser Klasse ist im Anhang B unter B.1 und B.2 aufgeführt. Zusätzlich wurde zum leichteren Verständnis auch der Quelltext der Klassen `Face`, `Point` und `Random` in den Anhang mit aufgenommen, da diese von der Klasse `Planet` genutzt werden. Auch bei den Klassen `Face`, `Point` und `Random` wurde auf Portabilität geachtet.

Der Konstruktor initialisiert im wesentlichen einige Variablen und legt den im Abschnitt 4.1.3 beschriebenen `seed`-Wert des Planeten fest. In der Methode `Solid()` wird dann der Grundkörper definiert, indem eine Liste von Eckpunkten (Instanzen der Klasse Point) und eine Liste von Seiten (Instanzen der Klasse Faces) aufgestellt wird. Da sich gezeigt hat, dass ein Ikosaeder die besten geometrischen Eigenschaften besitzt, wurde eins solcher als Grundkörper gewählt und

die in 2.3 berechneten Koordinaten verwendet.

Die Methode `CreatePoint()` berechnet für zwei gegebene Punkte einen neuen Punkt. Sie wird in der Methode `RefineFaces()` für je zwei Punkte eines Dreiecks aufgerufen, um es gemäß *Surface-Refinement* aufzuteilen. `CreatePoint()` hat als dritten Parameter noch einen Integerwert, der festlegt, in welchem Schritt sich der Algorithmus innerhalb der Planetenerzeugung befindet. Dies ist notwendig, da bei der Erzeugung eines neuen Punktes gleichzeitig seine Höhe bestimmt wird und sich Möglichkeiten des Zuwachses mit jedem Schritt verringern. Desweiteren wird in dieser geprüft, ob sich der neu generierte Punkt jenseits eines der beiden Polarkreise befindet.

Das in Abschnitt 4.1.2 erläuterte Level-of-Detail-Algorithmus wird dann im wesentlichen in den beiden Methoden `ReduceFaces()` und `RefineFaces()` implementiert. Beide arbeiten auf der Liste von Eckpunkten und der Liste der Seitenflächen. `ReduceFaces()` erhält als Parameter die Position der virtuellen Kamera. Zuerst wird über die Liste der Eckpunkte iteriert und wie in Abschnitt 4.1.2 beschrieben, der Eckpunkt gesucht, der den geringsten Abstand zur virtuellen Kamera aufweist. Ist dieser gefunden, wird über die Liste der Seitenflächen iteriert, um alle Flächen zu bestimmen, die diesen Punkt als Eckpunkt haben. Diese werden in einer neuen Liste gespeichert und die alte Liste gelöscht. Anschließend wird geprüft, welche Punkte zu den Seitenflächen in der neuen Liste gehören. Dazu muss zuerst über alle Eckpunkte aus der Eckpunkt-Liste und für jeden dieser Punkte über die Liste der Seitenflächen iteriert werden. Dies sieht auf den ersten Blick sehr aufwendig aus. Da aber beide Listen sehr kurz sind (die neue Liste der Seitenflächen enthält höchstens sechs, die Liste der Eckpunkte maximal 19 Einträge), fällt das Durchsuchen der Listen bei der Laufzeit des Programms nicht sehr ins Gewicht. Wird für einen Punkt festgestellt, dass er zu mindestens einer Seitenfläche gehört, so wird er in eine neue Punktliste aufgenommen, andernfalls gelöscht.

Die Methode `RefineFaces()` operiert ebenfalls auf den beiden Listen. Hier wird über alle Seitenflächen iteriert. Jede Seite wird zunächst aus der Liste entfernt und dann ihre die Eckpunkte bestimmt. Für jeweils zwei dieser Punkte wird dann mit Hilfe der Methode `CreatePoint()` ein neuer Punkt generiert und mit den drei alten und den drei neuen Punkten werden insgesamt vier neue Seitenflächen definiert und zur Liste der Seitenflächen hinzugefügt. Die neu erzeugten

Punkte werden zur Liste der Eckpunkte hinzugefügt.

In der Methode `GeneratePlanet()` wird zunächst jeweils eine Liste für die Punkte und die Seitenflächen des Planeten angelegt. Der anschließende Aufruf der Methode `Solid()` sorgt dafür, dass diese Listen mit den Werten für den Basiskörper gefüllt werden. Dann wird überprüft, ob mehr der Algorithmus mehr Schritte auszuführen hat, als die in der Konstante `DETAIL` festgelegte Anzahl von Schritten. Falls ja, wird die Methode `LOD()` ausgeführt. Hier werden `ReduceFaces()` und `RefineFaces()` wiederholt aufgerufen, um den darzustellenden Bereich einzuschränken.

Nachdem dies geschehen ist, wird für jede Seite aus der Seitenliste die Methode `Recursive()` ausgeführt. Diese erhält als Parameter die Eckpunkte der Seite, die Anzahl der verbliebenen Schritte und den aktuellen Schritt des Algorithmus. Dieser notwendig, da in `Recursive()`, wie bereits in der Methode `RefineFaces()`, mit Hilfe von `CreatePoint()` neue Punkte berechnet werden. Die Anzahl der verbliebenen Schritte dient dazu, das Ende der Rekursion zu bestimmen. Ist dies der Fall, so wird für jeden der Eckpunkte festgestellt, an welcher Position er gerendert werden soll (dazu wird die Methode `GetDisplacedPos()` aus der Klasse `Point` aufgerufen), der Terraintyp des Dreiecks bestimmt um es entsprechend einfärben zu können und schließlich die Methode `DrawTriangle()` der Klasse `GraphicCore` aufgerufen, die dann das Dreieck zeichnet. Ist das Rekursionsende noch nicht erreicht, wird das gegebene Dreieck, gemäß *Surface-Refinement* aufgeteilt und die Methode `Recursive()` für jedes der vier entstandenen Dreiecke aufgerufen.

Nach dem Aufruf von `Recursive()` in der Methode `GeneratePlanet()` werden nur noch alle Punkte und Seitenflächen des Planeten aus den entsprechenden Listen entfernt und gelöscht, um den belegten Speicher wiederfreizugeben.

Kapitel 5

Bewertung

In diesem Abschnitt werden die einzelnen Teile des entwickelten Verfahrens einer kritischen Analyse und Bewertung unterzogen. Außerdem werden die jeweiligen Vor- und Nachteile hervorgehoben und mögliche Ansätze zur Verbesserung aufgezeigt.

5.1 Verzerrungen beim Surface Refinement

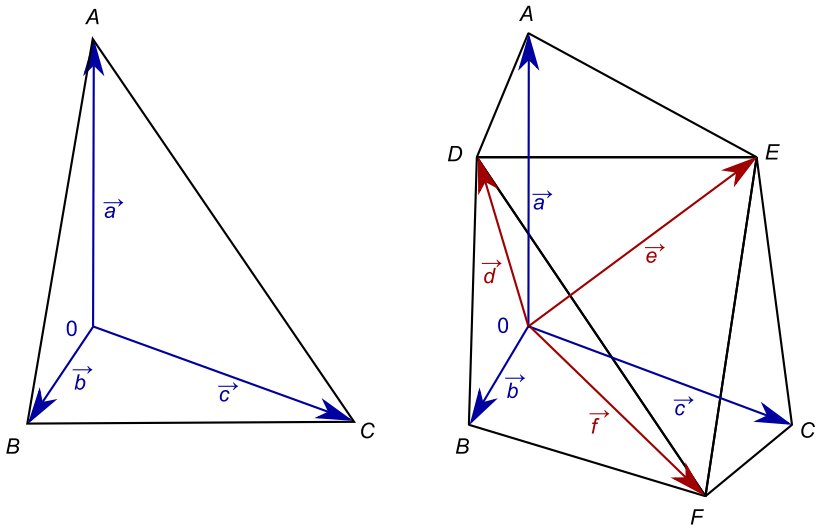

Abbildung 5.1: erster Schritt des Surface Refinement

Im Abschnitt 4.1.1 wurde mit *Surface-Refinement* ein Verfahren gezeigt, wie man aus einem Tetra-, Okta- oder Ikosaeder eine Polygondarstellung für eine Ku-

gel gewinnen kann. Die auf diese Weise erzeugten Kugeln weisen allerdings Unregelmäßigkeiten auf, da die Dreiecke nicht alle gleich groß sind. Ursache dafür ist die Normalisierung und anschließende Skalierung der neu berechneten Vektoren. Dadurch weisen diese Vektoren zwar die gleiche Länge, wie die bereits vorhandenen auf. Allerdings liegen sie nicht auf der Verbindungslinie der beiden Vektoren, aus dem sie berechnet wurden.

Die Abbildungen 5.1 und 5.2 verdeutlichen den Sachverhalt genauer. Abbildung 5.1 zeigt eine Seitenfläche des Tetraeders vor und nach dem ersten *Refinement*-Schritt. Abbildung 5.2 erläutert die in den folgenden Berechnungen benutzten Größen. Dabei sind die Punkte D' und E' die Mittelpunkte, der Strecken AB bzw. AC.

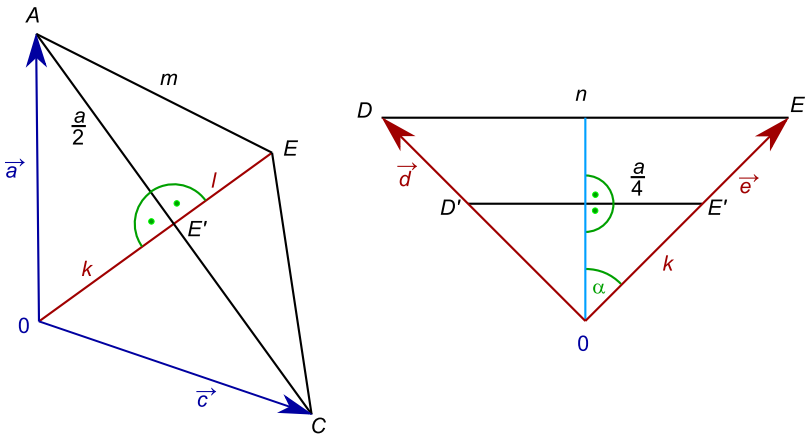

Abbildung 5.2: Kenngrößen zur Berechnung der Verzerrungen

Es gilt also:

$$k = \sqrt{r^2 - \frac{a^2}{4}} \text{ und } l = r - k = r - \sqrt{r^2 - \frac{a^2}{4}}$$

, da die Länge der Vektoren $\overrightarrow{a}$, $\overrightarrow{b}$ und $\overrightarrow{d}$ gleich dem Radius r entspricht. Daraus folgt also für m:

$$m = \sqrt{l^2 + \frac{a^2}{4}} = \sqrt{\left(r - \sqrt{r^2 - \frac{a^2}{4}}\right)^2 + \frac{a^2}{4}}$$

und für n:

$$\tan\alpha = \frac{\frac{a}{4}}{k} \text{ und } \frac{n}{2} = r \cdot \tan\alpha = r\frac{\frac{a}{4}}{\sqrt{r^2 - \frac{a^2}{4}}}$$

Wobei in den Abschnitten 2.1 bis 2.3 gezeigt würde, dass für jeden dieser Körper der Radius der Umkugel und die Seitenlänge der Begrenzungsflächen in einem eindeutigen Verhältnis stehen. In jedem Falle aber ist $m < n$.

Tetraeder	Oktaeder	Ikosaeder
$r = \frac{a}{4}\sqrt{6}$	$r = \frac{a}{2}\sqrt{2}$	$r = \frac{a}{4}\sqrt{10+2\sqrt{5}}$
$a = \frac{4}{\sqrt{6}}r$	$a = \frac{2}{\sqrt{2}}r$	$a = \frac{4}{\sqrt{10+2\sqrt{5}}}r$

Tabelle 5.1: Radius und Seitenlänge von Tetraeder, Oktaeder und Ikosaeder

In der Folge sind die neu entstandenen Dreiecke ADE, DBF und EFC nicht mehr gleichseitig, mehr noch, jeder Punkt ist Eckpunkt von sechs Dreiecken. Diese sollten jeweils ein regelmäßiges Sechseck bilden. Zu den Eckpunkten des Tetraeders werden diese aber immer stärker deformiert. Da die Differenz zwischen n und m gegen 0 geht, nimmt dieser Effekt zwar mit jedem Iterationsschritt ab und fällt nach wenigen Schritten nicht mehr auf.

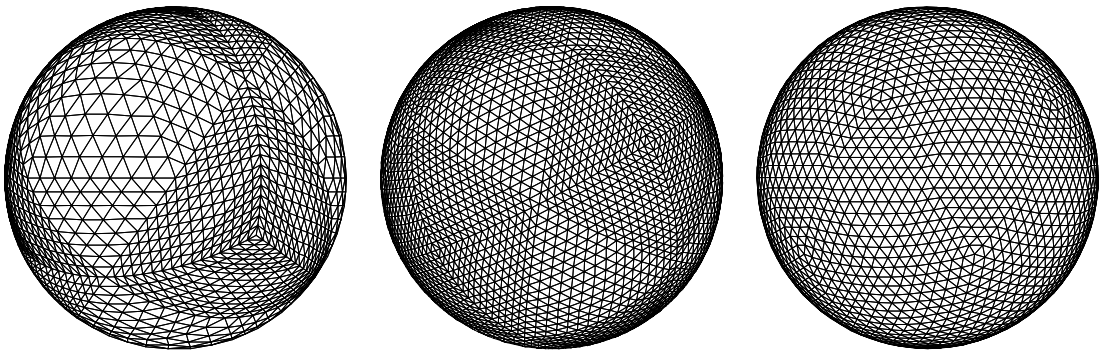

Abbildung 5.3: Verzerrungen am Tetraeder, Oktaeder und Ikosaeder

Wie Abbildung 5.3 zeigt, reichen aber die Verzerrungen der ersten Schritte aus, um deutlich sichtbare Unregelmäßigkeiten der Kugel zu verursachen. Der Tetraeder und der Oktaeder wurden mit fünf Schritten, der Ikosaeder mit vier Schritten berechnet. Man kann auf Abbildung 5.3 aber auch erkennen, dass dieser Effekt gemildert wird, wenn man statt eines Tetraeders einen Oktaeder verwendet. Beim Ikosaeder fallen die Verzerrungen kaum noch auf. Ursache dafür ist,

dass die Differenz zwischen m und n beim Tetraeder größer ist als beim Oktaeder, beim Ikosaeder ist sie am geringsten.

Es gibt allerdings noch eine weitere Auffälligkeit. Die bei jedem Schritt neu hinzugefügten Punkte sind Eckpunkte von sechs Dreiecken. Das gilt aber nicht für die Eckpunkte des Startkörper. So gehört jeder Eckpunkt eines Tetraeders zu drei, eines Oktaeders zu vier und eines Ikosaeders zu fünf Seitenflächen. Da das *Surface-Refinement* an diesem Umstand nichts ändert, sondern nur neue Punkte einfügt, gilt dies auch für die aus diesem Körper mit *Surface-Refinement* geschaffenen Kugeln. Da beim Iksoader diese Differenz am geringsten ist (fünf statt sechs Dreiecke), fällt es bei ihm auch am wenigsten auf, insbesondere dann, wenn die Punkte, wie im entwickelten Verfahren noch um einen zufälligen Höhenwert versetzt werden.

Wie gezeigt wurde, sind in beiden Fällen die verurachten Verzerrungen beim Tetraeder am größten und beim Ikosaeder am geringsten. Abschließend lässt sich also sagen, dass man beim *Surface-Refinement* die beste Darstellungsqualität erhält, wenn man als Ausgangskörper einen Ikosaeder nimmt. Aus diesem Grund wurde auch in der Referenzimplementierung ein Ikosaeder gewählt.

5.2 Der LOD-Teil

5.2.1 Vorteile

Der Vorteil des vorgestellten LOD ist, dass es sehr einfach und damit auch sehr schnell ist. Im ersten Schritt des LOD-Teils enthält eine Liste die Seiten des Ausgangskörpers und eine mit seinen Eckpunkten, im Falle eines Ikosaeders 20 Elemente in der Dreiecksliste und 12 Elemente in der Eckpunktliste. Hier wird bereits der dem Betrachter nächst gelegene Punkt und die mit ihm verbundenen Dreiecke ermittelt. Dazu sind 12 Vektorsubtraktionen durchzuführen und für deren Ergebnis anschließend der Betrag zu bestimmen. Ist der gesuchte Punkt ermittelt, müssen die Listen der Dreiecke und Eckpunkte abgesucht und die mit dem berechneten Punkt benachbarten Dreiecke und deren Eckpunkte in jeweils ein neue Liste übernommen werden. Die neue Dreiecksliste enthält nun fünf Dreicke[1] und die Eckpunktliste hat sechs Einträge. Nach dem rekursiven Aufteilen der

[1]Da jeder Eckpunkt des Ikosaeders 5 Seitenflächen hat.

Dreiecke gemäß *Surface-Refinement* vervierfacht sich die Dreiecksanzahl auf 20, die Anzahl der Eckpunkte erhöht sich auf 16. Wieder sind die entsprechenden Vektor- und Listenoperationen durchzuführen. Das Absuchen der Listen ist zwar teuer. Allerdings sind die Listen nur sehr kurz, denn im i. Schritt des LOD-Teils startet ein LOD-Schritt mit maximal sechs Dreiecken und sieben Eckpunkten und erhält nach dem Surface Refinement 24 Dreiecke und 19 Eckpunkte wie man Abbildung 5.4 entnehmen kann.

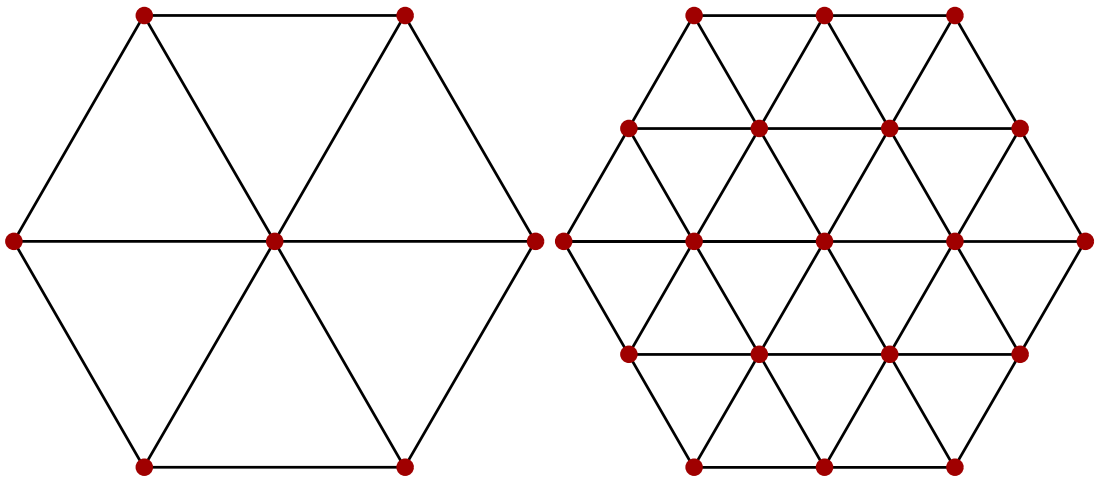

Abbildung 5.4: Dreiecksnetz vor und nach einem Refinement-Schritt

Davon werden wiederum maximal sechs Dreiecke und sieben Eckpunkte ausgewählt, mit denen der $i+1$. Schritt des LOD-Teils startet. Es sind also in jedem LOD-Schritt maximal 19 Vektorsubtraktionen durchzuführen und die Länge deren Ergebnisses zu bestimmen[2]. Anschließend sind maximal 24 Einträge in der Liste der Dreiecke und 19 Einträge in der Liste der Eckpunkte zu überprüfen. Aber hier kann man die Suche vorzeitig abbrechen, wenn man bereits sechs Dreiecke und sieben Eckpunkte gefunden hat. Denn jeder Punkt in einem solchen Dreiecksnetz ist Teil von höchstens sechs Dreiecken und diese haben zusammen sieben Eckpunkte (vgl. Abbildung 5.4).

[2]Wobei das Wurzelziehn bei der Ermittlung des Betrages eines Vektors die aufwendigste Operation darstellt

5.2.2 Nachteile

Leider hat dieser LOD-Algorithmus aber gerade wegen seiner Schlichtheit auch einige gravierende Nachteile. Wie alle LOD-Algorithmen leidet auch dieses unter dem sogenannten LOD popping (siehe dazu auch Abschnitt 3.1). Beim Übergang zwischen zwei Detailstufen werden zusätzliche Dreiecke hinzugefügt oder entfernt, was zu einer mehr oder weniger starken Veränderung des Bildes führt. Im Allgemeinen ist dieser Effekt für den Betrachter sichtbar und wird als störend empfunden. Man kann ihn aber abmildern, wenn beim Übergang von einer Detailstufe zur anderen langsam gemorpht wird bzw. Dreiecke über mehrere Frames ein- oder ausgeblendet werden.

Nun macht der LOD-Schritt nichts anderes, als den Standort des Betrachters zu bestimmen und um ihn herum ein sechseckiges Areal mit einer festgelegten Anzahl von Polygonen zu zeichnen. Dies vereinfacht (und beschleunigt damit) die Berechnung. Ein guter LOD-Algorithmus orientiert sich aber daran, was der Betrachter sieht und was nicht. So werden im LOD-Schritt z. B. alle Polygone rund um den Betrachter berechnet, auch wenn er diejenigen, die hinter ihm liegen, nicht zu sehen bekommt. Andererseits kann es passieren, dass der Betrachter über den Radius des vom LOD-Schritt berechneten Areals hinausblicken kann. Freilich kann man dieses Problem kaschieren, indem man Nebel in die Szene einfügt, so dass alles, was über den gerenderten Bereich hinausgeht, vom Nebel verdeckt wird. Dennoch bleibt dies ein grundlegendes Problem und der LOD-Teil ist ohne Frage verbesserungswürdig und -fähig.

Ein mit ROAM oder Röttger vergleichbares Verfahren würde hier natürlich die besten Ergebnisse erzielen. Leider lassen sich diese, wie schon in Abschnitt 4.1.2 gezeigt, nicht einfach mit *Surface-Refinement* kombinieren. Hier müsste ein vergleichbares Verfahren entwickelt werden, das auf die gegebenen Besonderheiten (insbesondere gleichseitige Dreiecke als Seitenflächen) eingeht.

5.2.3 Verbesserungsvorschläge

Es wurden während der Diplomarbeit eine ganze Reihe von Ideen untersucht, um LOD-Teil zu verbessern. Leider wiesen sie selbst wiederum Nachteile auf bzw. lassen sich nur unter bestimmten Voraussetzungen nutzen. Einge Verbesserungsvorschläge werden im folgenden kurz vorgestellt.

Eine Möglichkeit, das vorgestellte Verfahren zu verbessern, ergibt sich, wenn man die Bewegungsmöglichkeiten des Betrachters auf die Planetenoberfläche beschränkt und davon ausgeht, dass er sich in seiner Umgebung nicht mit allzugroßer Geschwindigkeit bewegt. In diesem Fall ist Anzahl der LOD-Schritte mehr oder weniger konstant und ändert sich nur langsam, wenn man davon ausgeht, dass sich Höhenwerte in einem gegebenen Areal nicht allzu drastisch verändern. In diesem Fall könnte man beispielsweise den LOD-Teil etwas abwandeln und die berechneten Geometriedaten cachen.

So wäre es zum Beispiel denkbar, im LOD-Teil beim letzten Schritt nicht nur dem zum Betrachter nächsten Punkt und die mit ihm verbundenen Dreiecke auszuwählen, sondern zusätzlich zu bestimmen, in welchem dieser Dreiecke er genau steht. Für jeden Eckpunkt dieses Dreiecks werden alle benachbarten Dreiecke bestimmt. Dies werden dann rekursiv unterteilt und das Ergebnis gecacht. Nun muss nur noch der LOD-Teil ausgeführt werden um zu sehen, welcher dieser drei Punkte dem Betrachter am nächsten ist und das gecachte Ergebniss für diesen angezeigt werden. Verlässt der Betrachter das Dreieck, für das bereits die gecachten Werte vorliegen, muss der Vorgang für die Eckpunkte des Dreiecks durchgeführt werden, indem sich der Betrachter nun befindet. Da aber das alte und das neue Dreieck benachbart sind, haben sie im allgemeinen eine gemeinsame Seite[3] und damit auch zwei gemeinsame Eckpunkte für die bereits die Ergebnisse berechnet wurden. Effektiv muss die rekursive Berechnung also nur für einen neuen Punkt durchgeführt werden. Neben den obengenannten Restriktionen hat diese Alternative auch den Nachteil, dass der rekursive Teil sehr häufig ausgeführt werden muss, wenn der Betrachter häufig zwischen zwei solcher Dreiecke hin und her wechselt.

Eine anderer Weg zur Steigerung der Effizienz wäre, im letzten LOD-Schritt zusätzlich zur Betrachterposition auch dessen Blickrichtung zu bestimmen. Dann könnte man alle Dreiecke, die hinter ihm liegen, ebenfalls aus der Liste streichen und würde den rekursiven Teil lediglich auf die verbliebenen anwenden. Allerdings bietet dieses Vorgehen nur dann einen wirklichen Vorteil, wenn der Betrachter von seinen Standpunkt aus nach außen und nicht ins Zentrum des berechneten Areals

[3]Es ist möglich, wenn auch sehr unwahrscheinlich, dass der Betrachter in ein direkt gegenüberliegendes Dreieck übergeht ohne dass das alte und das neue Dreieck eine gemeinsame Kante haben. In diesem Fall müßten der rekursive Teil zweimal ausgeführt werden.

blickt.

Abschließend lässt sich sagen, dass der entwickelte LOD-Algorithmus einfach und dadurch sehr schnell ist, aber auch keine sehr guten Ergebnisse liefert. Hier ist ein Verfahren notwendig, das wie ROAM oder Röttger Geländemerkmale bzw. Bildfehler einsetzt, um Polygonzahl zusätzlich zu verringern.

5.3 Die fraktale Modellierung

Wie schon gezeigt, wurde für die Landschaftsmodellierung auf eine abgewandelte Form des Plasma-Algorithmus zurückgegriffen. Warum nicht Perlin Noise? Möchte man den Planeten in einer sehr hohen Auflösung darstellen, sind bei Perlin Noise sehr viele Frequenzen aufzusummieren. Dadurch wird das Verfahren aber sehr aufwendig und langsam, auch wenn Ken Perlin unter [17] eine Möglichkeit gefunden hat, diesem Problem zu begegnen. Dennoch wurde Perlin Noise als Grundlage verworfen, da der Aufwand einfach zu groß ist und im Vergleich zu Midpoint Displacement keine signifikanten Qualtitätsverbesserung bringt.

Midpoint Displacement bietet zwei Varianten Diamond-Square und Plasma, wobei die Wahl auf letzteren fiel. Zwar weist Plasma, wie in Abschnitt 3.2.2 gezeigt, gegenüber dem Diamond-Square-Algorithmus Schwächen auf, dennoch wurde diese Methode gewählt, da sie einfach zu berechnen ist. Es lässt sich zwar auch eine Diamond-Square-ähnliche Variante einsetzen, diese ist jedoch deutlich langsamer. Es wurde gezeigt, dass beim Diamond-Square-Algorithmus, im Gegensatz zu Plasma, bei der Interpolation der Höhenwerte der Seiten nicht nur deren Eckpunkte in die Berechnung eingehen, sondern auch die Höhenwerte der Mittelpunkte der benachbarten Quadrate (siehe hierzu noch einmal Abbildung 3.14). Eine vergleichbare Vorgehensweise lässt sich auch hier einsetzen. Allerdings ist dazu bereits beim *Surface-Refinement*, bei jedem Refinement-Schritt anzusetzen. Im Allgemeinen liegt jeder Punkt P, der neu hinzugefügt werden soll, genau in der Mitte der Kante zweier Punkte A und B. Diese gehören ihrerseits zu zwei Dreiecken ABC und ABD (siehe Abbildung 5.5). In diesem Fall wird die neue Höhe von P nicht nur aus den Punkten A und B interpoliert, wie es bei Plasma der Fall ist, sondern sie ergibt sich als Mittelwert der Punkte A, B, C und D.

Damit lässt sich ein mit Diamond-Square vergleichbares Verhalten und Resultat erreichen. Abbildung 5.6 zeigt, wie dies beim *Surface-Refinement* in ei-

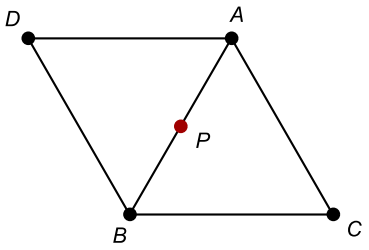

Abbildung 5.5: „Diamond-Step„

nem Dreiecksnetz aussehen könnte. Wie gezeigt, ist es bei dieser Vorgehensweise von zentraler Bedeutung, dass für jede Dreickskante bestimmt wird, zu welchen beiden Dreiecken sie gehört. Solche Nachbarschaftstests sind aber in der Regel aufwendig, besonders, wenn die Dreiecke in einer Liste organisiert sind. Unter Umständen ist dann nämlich die gesamte Liste abzusuchen und für jedes Dreieck zu prüfen, ob es die gesuchten Punkte A und B enthält. Auch eine Organisation der Dreiecke in einer Baumstruktur bringt keine Vorteile, da für manche Kanten die Suche von der Wurzel aus gestartet werden muss.

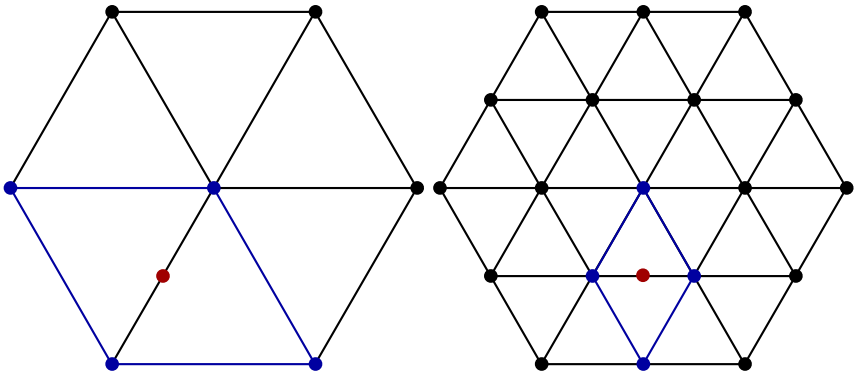

Abbildung 5.6: „Diamond-Steps“ im Dreiecksnetz

Auch wenn die mit Hilfe von fraktalen Methoden erzeugten Landschaften auf den ersten Blick sehr gut aussehen, fehlen ihnen doch Eigenschaften, die jede natürliche Landschaft aufweist - Erosion, Vulkanismus und Spuren anderer, landschaftsverändernder Prozesse. Auf jede Landschaft wirken vielfältige Faktoren wie Wind, Regen, Gletscher, Vukan, Erdbeben usw. ein und verändern diese. Die Folgen dieser Vorgänge lassen sich jedoch mit einem einfachen Modell, wie es die fraktalen Methoden bieten, nicht realisieren. Musgrave zeigt aber bereits in seiner Dissertation [2] Wege auf, Errosion und weitere Effekte in die Erzeugung

mit einzubeziehn, um realistischere Landschaften zu erstellen.

Auch ist es mit den vorgestellten Methoden nicht möglich, Flüsse nachzubilden und die Veränderungen, die sie in realen Gelände verursachen. Einen Grand Canyon oder ein Nildelta wird man in solchen virtuellen Landschaften vergebens suchen.

Bei derartigen Überlegungen taucht natürlich die Frage auf, ob und wie weit sich die Erzeugung eines Planeten manipulieren lässt, um sein Erscheinungsbild zu beeinflussen. Überlässt man die Entstehung des Planeten dem Zufall, das heißt dem *seed*-Wert, hat man keinerlei Kontrolle. Allerdings kann man die „Zufallsfunktion“ so verändern, dass sie für bestimmte Koordinaten vorher festgelegte Höhenwerte berechnet. Auf diese Art und Weise kann man die Form, die der Planet im Laufe der Berechnung annimmt, beeinflussen und in einem gewissen Rahmen steuern.

Dieser Form der Manipulation sind jedoch enge Grenzen gesetzt. So kann man zwar im Prinzip für jeden beliebigen Punkt die Höhe vorweg festlegen, allerdings wirkt sich die festgelegte Höhe nur für diese Koordinaten und in der laufenden Berechnung noch zu ermittelnde Koordinaten aus. Die Höhe benachbarter Punkte, die in vorhergehenden Berechnungsschritten des Algorithmus bestimmt wurden, ist von dieser erzwungen Höhe völlig unabhängig. Es können tiefe Löcher bzw. hohe Spitzen entstehen, falls die vorgegebenen Höhe stark von der Höhe der bereits berechneten Koordinaten abweicht.

Diese Form des „deterministischen Modellierens“ macht also nur Sinn, wenn man sich auf die ersten Schritte der Berechnung beschränkt und dann die Werte für alle Koordinaten eines Schrittes festlegt.

5.4 Parallelität

Der LOD-Teil des Algorithmus arbeitet in seiner Berechnung sehr sequentiell, denn für jeden Schritt muss das Ergbenis des vorhergehenden Schrittes vorliegen. Das ist aber nicht weiter problematisch, da jeder LOD-Schritt mit wenigen Berechnungen auskommt und dieser Teil des Algorithmus schnell abgearbeitet ist. Dagegen weist der rekursive Teil einen hohen Grad Parallelität auf. Der Grund dafür ist die rekursive Struktur dieses Abschnitts der Berechnung. Für jedes Dreieck hängt die weitere Berechnung bzw. Verarbeitung nur von diesem selbst (ge-

nauer gesagt von seinen Koordinaten und der Rekursionstiefe) ab. Die Ergebnisse benachbarter oder gar weiter entfernter Dreiecke spielen keine Rolle und müssen nicht berücksichtigt werden. Diese Unabhängigkeit in der Berechnung lässt sich nun nutzen, um jeden Schritt im rekursiven Teil in einen eigenen Thread oder Prozess auszulagern und diese wiederum auf verschiedene Prozessoren zu verteilen. Damit lässt sich dieser rechenintensive[4] Teil des Verfahrens auf Rechencluster verteilen und sich so eine noch höhere Darstellungsqualität erreichen.

Es muss aber nicht gleich ein Cluster sein. Gerade zur Zeit befindet sich die Computertechnologie in einer Umbruchphase. Da die Leistungssteigerung der Prozessoren nicht mehr beliebig über eine Steigerung der Taktfrequenz erreicht werden kann (Abwärme und Verlustleistung sind bereits heute große Probleme und werden in immer stärkerem Maße die Entwicklung hemmen), wird heute zunehmend auf parallele Architekturen gesetzt. So sind Mehrprozessorsysteme bei Severn und Highend-Workstations schon seit langem üblich. Mit Technologien wie z. B. Hyperthreading und MultiCore-Architekturen ziehen sie aber zunehmend auch in Desktopsysteme ein. So bestitzt der Athlon 64 X2 von AMD einen Prozessor mit zwei Kernen, während Intel mit dem Pentium D und dem Pentium Extreme Edition 840 ebenfalls einen Prozssor mit Doppelkern anbietet, wobei letzerer zusätzlich Hyperthreading unterstützt, somit also vier virtuelle Prozessoren bietet. Der Power Mac G5 Quad von Apple bietet dagegen vier echte Kerne, er ist mit zwei PowerPC G5 Dual-Core-Prozessoren von IBM ausgestattet.

Diese Entwicklung macht aber vor CPUs nicht Halt. So bieten sowohl Nvidia als ATI inzwischen die Möglichkeit zwei Grafikkarten zu kombinieren und ihre Leistung zusammen zu nutzen, was theorethisch die doppelte Grafikleistung verspricht. Der nächste Schritt dürfte wohl auch hier die Verwendung von zwei oder später sogar mehr Kernen auf einer Grafikkarte sein.

Mit der Verwendung von MultiCore-Architekturen steigt auch die Bedeutung von parallelen Algorithmen, denn nur sie können diese Technologien effizient nutzen. Das hier entwickelte Verfahren weist diese Parallelität auf und ist somit auf zukünftige Entwicklungen bestens vorbereitet.

[4]Rechenintensiv, weil in diesem Teil die Anzahl der Dreiecke exponentiell wächst.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Wie die Diplomarbeit gezeigt hat, lassen sich zwar ganze Planeten ohne größere Verzerrungen mit Hilfe fraktaler Methoden modellieren. Allerdings stößt die Darstellungsqualität an ihre Grenzen, da sich gängige Level-of-Detail-Algorithmen, wie ROAM bzw. Röttger, nicht einfach an die durch das Surface-Refinement gegebenen Bedingungen anpassen. Insbesondere die Triangulierung durch gleichseitige Dreiecke hat sich als problematisch erwiesen. Ohne diese LOD-Techniken kann aber nur eine relativ geringe Auflösung berechnet werden.

Der vorgestellte Level-of-Detail-Algorithmus stellt zwar keinen Ersatz für die obengenannten Verfahren dar. Er bietet aber eine sehr gute Grundlage, denn er schränkt einfach und dadurch sehr schnell den Bereich ein, in dem sich der Betrachter bzw. die virtuelle Kamera befindet. Dies ist vorallem auch deshalb wichtig, weil die bisher entwickelten LOD-Algorithmen nur mit vergleichsweise kleinen Flächen wirklich effizient funktionieren. Eine Kombination aus dem in der Diplomarbeit entwickelten Verfahren und einem ROAM/Röttger-ähnlichen Algorithmus würde deren jeweiligen Schwächen beheben.

Die eigentliche Modellierung der Landschaft bzw. der Gebirgszüge lässt sich dagegen problemlos auch auf sphärische Körper übertragen, zumindest wenn man dafür den Plasma-Algorithmus verwendet.

6.2 Ausblick

Wie bereits erwähnt, bietet auch Raytracing die Möglichkeit, mit fraktal-modellierten Landschaften zu rendern. Programme wie Bryce und Terragen erzeugen dabei Bilder von erstaunlicher Qualität. Besonders Licht- und Schatteneffekte (z. B. Reflexionen an der Wasseroberfläche) tragen sehr zu Immersion und Realismus bei. Leider ist Realtime Raytracing immer noch großen Clustern bzw. kleinen Szenen vorbehalten. Allerdings findet auf diesem Gebiet eine rege Entwicklung und Forschung statt. Besonders die Computer-Grafik-Gruppe um Professor Philipp Slusallek an der Universität Saarbrücken hat die Arbeit an Realtime Rendering vorangetrieben. Dabei sind die Projekte OpenRT [20] und SaarCOR [21] von besonderer Bedeutung. OpenRT ist eine sehr effiziente Implementierung des Raytracing Algorithmus mit einer zu OpenGL vergleichbaren API. Bei SaarCOR handelt es sich um eine vollständig programmierbare Raytracing Hardware Architektur, die mit der GPU heutiger Grafikkarten verglichen werden kann. Obwohl bisher von dieser Architektur nur FPGA-Prototypen existieren, die noch dazu mit moderaten 66 MHz laufen, sind die Ergebnise beeindruckend. Das frühe Entwicklungsstadium und die Tatsache, dass die Architektur sehr gut skaliert, geben dieser Technologie ein großes Potenzial, so dass sich Realtime Raytracing zu einer brauchbaren Alternative im Bereich der Darstellung von Computergrafik entwickeln kann.

Zwar ist der Rechenaufwand bei Raytracing wenig oder kaum von der Anzahl der Polygone einer Szene abhängig. Allerdings bringt die Visualisierung ganzer Planeten in einer hohen Auflösung ein enormes Datenvolumen mit sich, das schnell die Grenzen auch zukünfiger Computerarchitekturen überfordert. Die Polygonzahl vervierfacht sich in jedem Verfeinerungsschritt, wächst also exponentiell. Auch hier kann das vorgestellte LOD eingesetzt werden, um die Menge der Daten schnell auf ein handhabares Maß zu Reduzieren. Auf alle Fälle wird aber das Raytracing im Bereich der Landschaftsvisualisierung in Zukunft eine noch größere Rolle spielen und auch verstärkt in Echtzeitanwendungen Einzug halten. Das im Rahmen der Diplomarbeit entwickelte Verfahren kann auch hier zur weiteren Steigerung der Effizienz eingesetzt werden.

Anhang A

Screenshots

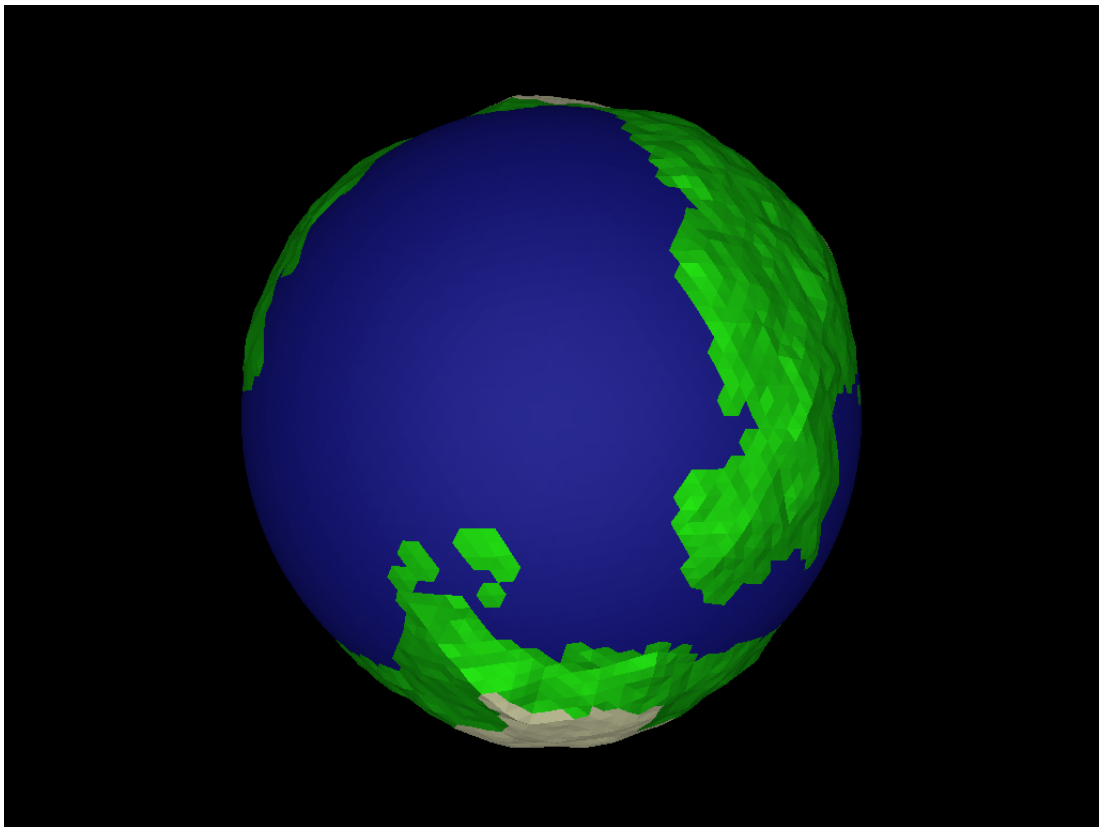

Abbildung A.1: Planet mit seed-Wert 1006 mit 5 vollständigen Rekursionsschritten

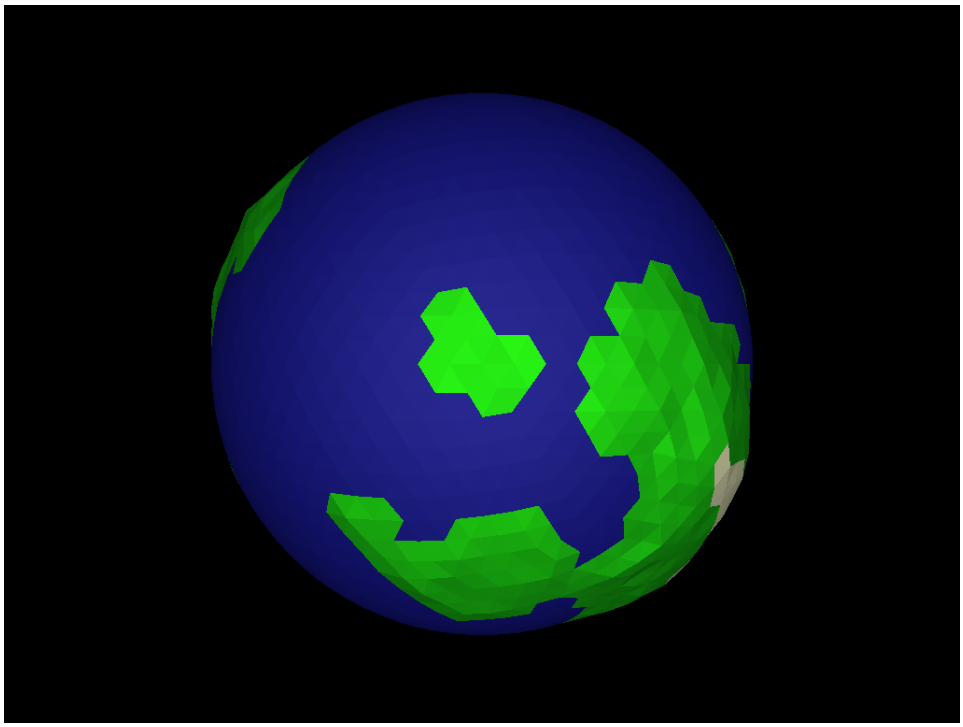

Abbildung A.2: Planet mit seed-Wert 666 mit 4 vollständigen Rekursionsschritten

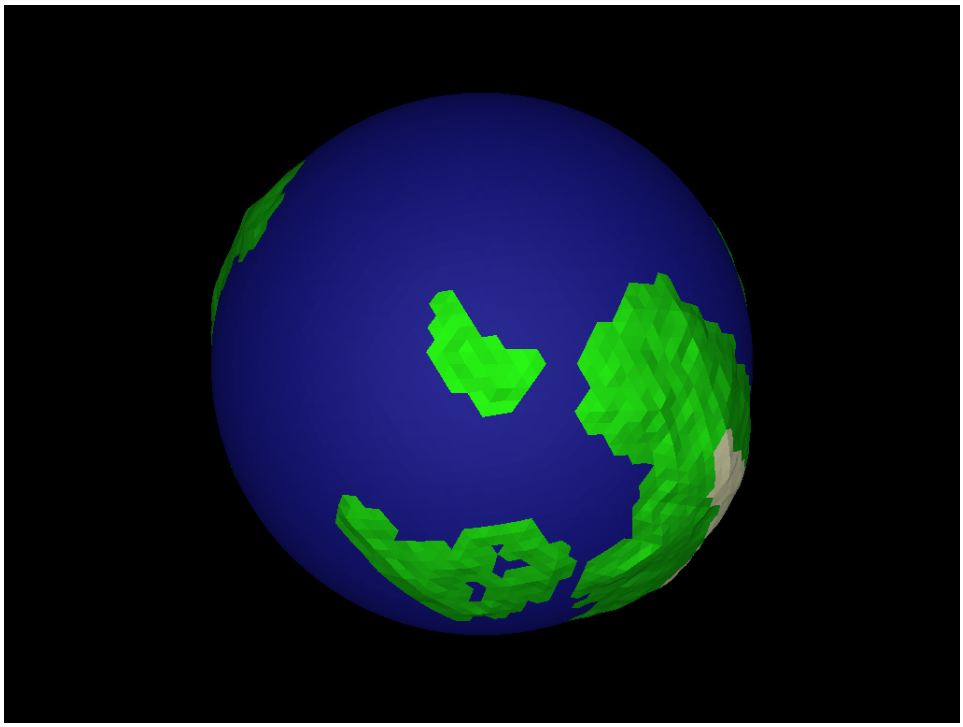

Abbildung A.3: Planet mit seed-Wert 666 mit 5 vollständigen Rekursionsschritten

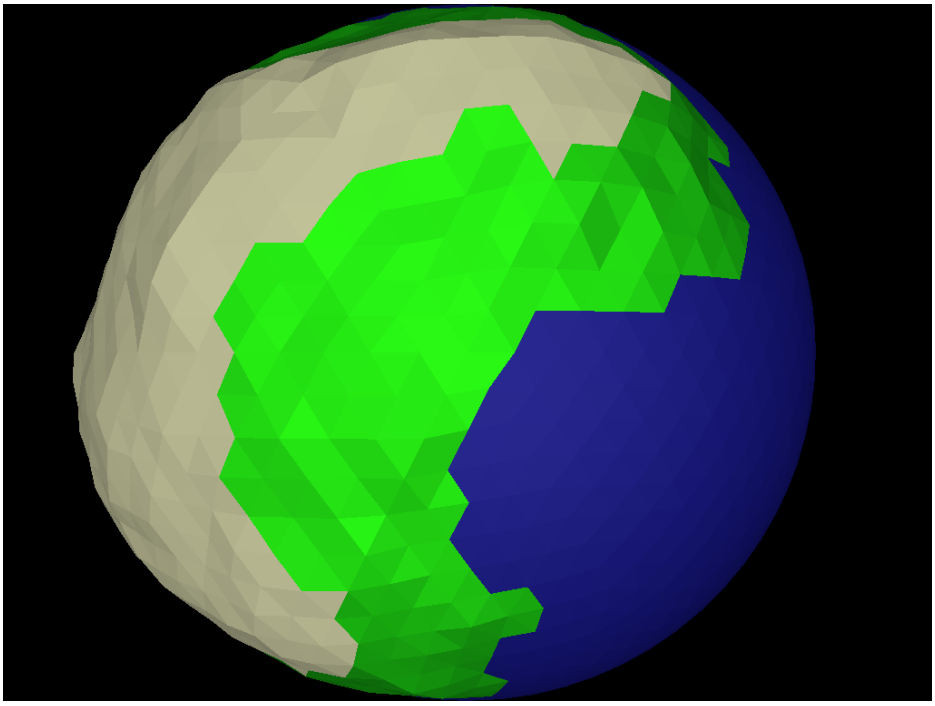

Abbildung A.4: Planet mit seed-Wert 1004 mit 4 vollständigen Rekursionsschritten

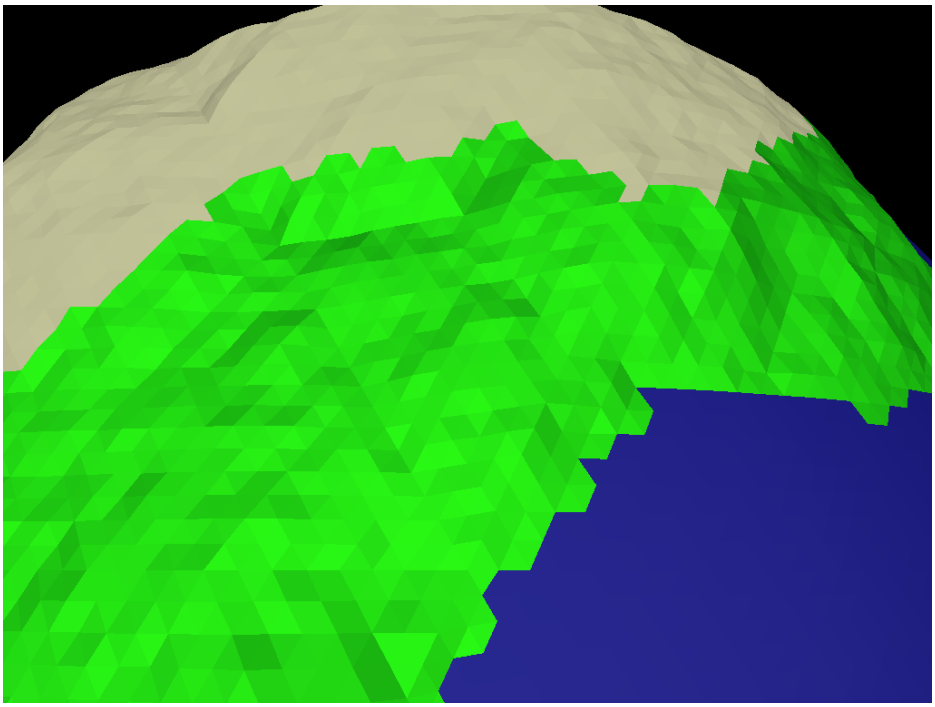

Abbildung A.5: Planet mit seed-Wert 1004 aus einem niedrigeren Betrachtungswinkel

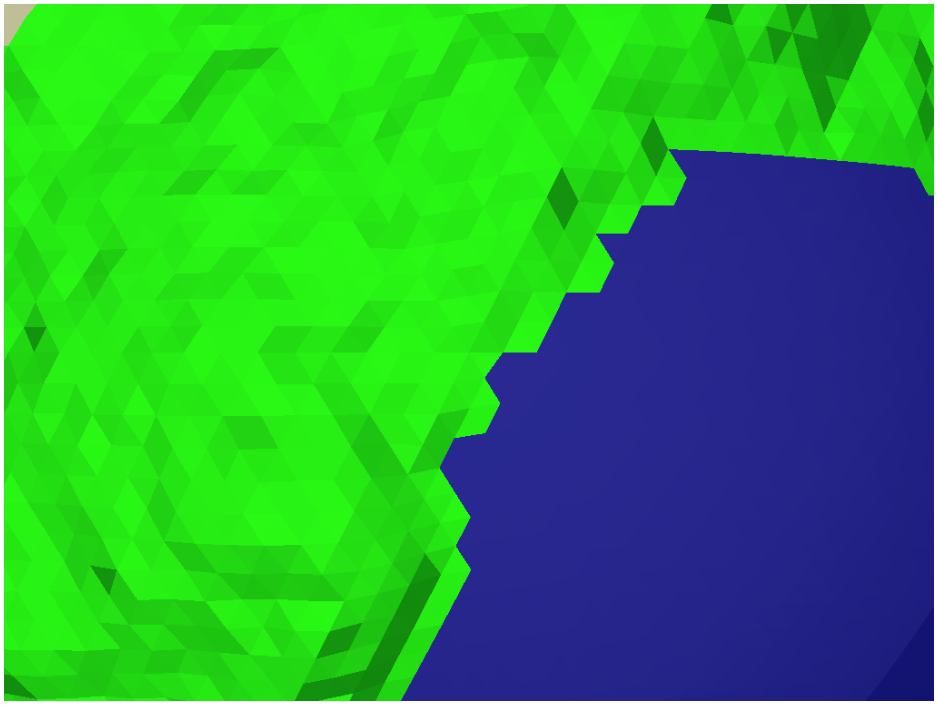

Abbildung A.6: Planet mit seed-Wert 1004 aus einem niedriger Höhe und hohem Detailgrad

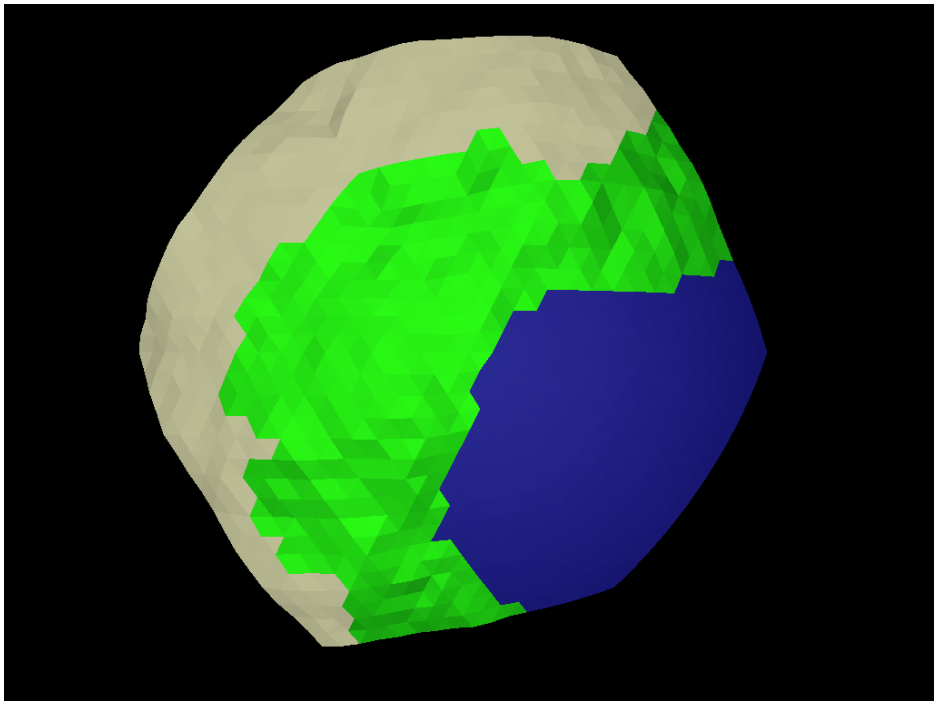

Abbildung A.7: gleicher Planetenausschnitt wie Abbildung A.6, aber aus größerer Höhe

Anhang B

Quellcode

Hier wird der Quellcode für die wichtigsten Klassen (`Planet`, `Face`, `Point` und `Random`) aufgelistet. Einige Methoden dieser Klassen werden in Abschnitt 4.2 näher erläutert.

B.1 Planet.h

```
#include <vector>
#include <list>
#include <climits>
#include <cmath>

#include "TypeDefs.h"
#include "GraphicCore.h"

#include "Point.h"
#include "Face.h"

#include "Random.h"

//symbolic infinity
#define INFINITY ULONG_MAX
//the number of the fully recursive steps
#define DETAIL 4
//this means the height can vary by 10 % of planet radius
#define HEIGHT_VARIANCE 0.1f
//the y-value of the polar circle
#define POLAR_CIRCLE 0.917f

//some terrain type constants
```

```
#define TERRAIN_WATER 0
#define TERRAIN_GRASS 1
#define TERRAIN_ROCK 2
#define TERRAIN_ICE 3


class Planet
{
private:

  int Seed; //the seed value determines the appearance
  int MaxSteps; //number of the refinement steps performed
  int ActualStep; //the actual step of the refinement

  Random Rand; //instance of the Random class

  std::list<Point*>* Points; //list points resp vertices
  std::list<Point*>::iterator PointIterator;

  std::list<Face*>* Faces; //list of faces
  std::list<Face*>::iterator FaceIterator;

  GraphicCore GC; //instance of the GraphicCore class

public:
  //constructor for initialisation of values
  Planet(void);
  //defines the vertices and the faces of an icosahedron
  void Solid(void);
  //computes the height scale factor for a given point and step
  float HeigtOffset(int step, Point* P);
  //creates a new point between the given two points
  Point* CreatePoint(int, Point*, Point*);
  //computes the terrain type for a face from the properties
  //of it's vertices
  char ComputeTerrain(Point*, Point*, Point*);
  //refines all faces of the faces list
  void RefineFaces(void);
  //removes faces from the faces list according to the cam pos
  void ReduceFaces(Vector*);
  //performs the LOD part of the algorithm
  void LOD(Vector*);
  //generates and draws the planet fully or partially
```

```
  void GeneratePlanet(Vector*, int);
  //the fully recursive refinement part
  void Recursive(Point*, Point*, Point*, int, int);
  //adds a point to the point list
  void AddPoint(Point*);
  //adds a face to the face list
  void AddFace(Face*);
  //destructor of the class
  ~Planet(void);
};
```

B.2 Planet.cpp

```
#include "Planet.h"

/*****************************************************************
 * The constructor of the planet. Initializes the steps and the *
 * seed of the planet which determines it's appearance.         *
 *****************************************************************/
Planet::Planet(void)
{
  Seed=666;
  MaxSteps=0;
  ActualStep=0;
}

/*****************************************************************
 * Initialisation of the planet. The base solid is defined      *
 * (here a icosahedron).                                        *
 *****************************************************************/
void Planet::Solid(void)
{
  //create the 16 vertices of the icosahedron
  Point* IcoPointA = new Point();
  Point* IcoPointB = new Point();
  Point* IcoPointC = new Point();
  Point* IcoPointD = new Point();
  Point* IcoPointE = new Point();
  Point* IcoPointF = new Point();
  Point* IcoPointG = new Point();
  Point* IcoPointH = new Point();
  Point* IcoPointI = new Point();
```

```
Point* IcoPointJ = new Point();
Point* IcoPointK = new Point();
Point* IcoPointL = new Point();

//define vertices for the icosahedron
(*IcoPointA).SetPos(+0.00000000f, +1.00000000f, +0.00000000f);
(*IcoPointB).SetPos(-0.27639320f, +0.44721360f, +0.85065081f);
(*IcoPointC).SetPos(+0.72360680f, +0.44721360f, +0.52573111f);
(*IcoPointD).SetPos(+0.27639320f, -0.44721360f, +0.85065081f);
(*IcoPointE).SetPos(+0.89442719f, -0.44721360f, +0.00000000f);
(*IcoPointF).SetPos(+0.72360680f, +0.44721360f, -0.52573111f);
(*IcoPointG).SetPos(+0.27639320f, -0.44721360f, -0.85065081f);
(*IcoPointH).SetPos(-0.27639320f, +0.44721360f, -0.85065081f);
(*IcoPointI).SetPos(-0.72360680f, -0.44721360f, -0.52573111f);
(*IcoPointJ).SetPos(-0.89442719f, +0.44721360f, +0.00000000f);
(*IcoPointK).SetPos(-0.72360680f, -0.44721360f, +0.52573111f);
(*IcoPointL).SetPos(+0.00000000f, -1.00000000f, +0.00000000f);

//set the north and the south pole
(*IcoPointA).SetPolarZone(true);
(*IcoPointL).SetPolarZone(true);

//compute the heigt scale factors of icosahedron veritzes
(*IcoPointA).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointA));
(*IcoPointB).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointB));
(*IcoPointC).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointC));
(*IcoPointD).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointD));
(*IcoPointE).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointE));
(*IcoPointF).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointF));
(*IcoPointG).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointG));
(*IcoPointH).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointH));
(*IcoPointI).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointI));
(*IcoPointJ).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointJ));
```

```
(*IcoPointK).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointK));
(*IcoPointL).SetHeightScaleFactor(1.0f +
                                  HeigtOffset(0, IcoPointL));

//displace the vertices
(*IcoPointA).DisplacePoint();
(*IcoPointB).DisplacePoint();
(*IcoPointC).DisplacePoint();
(*IcoPointD).DisplacePoint();
(*IcoPointE).DisplacePoint();
(*IcoPointF).DisplacePoint();
(*IcoPointG).DisplacePoint();
(*IcoPointH).DisplacePoint();
(*IcoPointI).DisplacePoint();
(*IcoPointJ).DisplacePoint();
(*IcoPointK).DisplacePoint();
(*IcoPointL).DisplacePoint();

//create the 20 faces of the icosahedron
Face IcoFace[20];
Face* IcoFace01 = new Face();
Face* IcoFace02 = new Face();
Face* IcoFace03 = new Face();
Face* IcoFace04 = new Face();
Face* IcoFace05 = new Face();
Face* IcoFace06 = new Face();
Face* IcoFace07 = new Face();
Face* IcoFace08 = new Face();
Face* IcoFace09 = new Face();
Face* IcoFace10 = new Face();
Face* IcoFace11 = new Face();
Face* IcoFace12 = new Face();
Face* IcoFace13 = new Face();
Face* IcoFace14 = new Face();
Face* IcoFace15 = new Face();
Face* IcoFace16 = new Face();
Face* IcoFace17 = new Face();
Face* IcoFace18 = new Face();
Face* IcoFace19 = new Face();
Face* IcoFace20 = new Face();

//define faces for the icosahedron
```

```
(*IcoFace01).SetFacePoints(IcoPointA, IcoPointB, IcoPointC);
(*IcoFace02).SetFacePoints(IcoPointA, IcoPointC, IcoPointF);
(*IcoFace03).SetFacePoints(IcoPointA, IcoPointF, IcoPointH);
(*IcoFace04).SetFacePoints(IcoPointA, IcoPointH, IcoPointJ);
(*IcoFace05).SetFacePoints(IcoPointA, IcoPointJ, IcoPointB);
(*IcoFace06).SetFacePoints(IcoPointB, IcoPointD, IcoPointC);
(*IcoFace07).SetFacePoints(IcoPointC, IcoPointD, IcoPointE);
(*IcoFace08).SetFacePoints(IcoPointC, IcoPointE, IcoPointF);
(*IcoFace09).SetFacePoints(IcoPointF, IcoPointE, IcoPointG);
(*IcoFace10).SetFacePoints(IcoPointF, IcoPointG, IcoPointH);
(*IcoFace11).SetFacePoints(IcoPointH, IcoPointG, IcoPointI);
(*IcoFace12).SetFacePoints(IcoPointH, IcoPointI, IcoPointJ);
(*IcoFace13).SetFacePoints(IcoPointJ, IcoPointI, IcoPointK);
(*IcoFace14).SetFacePoints(IcoPointJ, IcoPointK, IcoPointB);
(*IcoFace15).SetFacePoints(IcoPointB, IcoPointK, IcoPointD);
(*IcoFace16).SetFacePoints(IcoPointD, IcoPointL, IcoPointE);
(*IcoFace17).SetFacePoints(IcoPointE, IcoPointL, IcoPointG);
(*IcoFace18).SetFacePoints(IcoPointG, IcoPointL, IcoPointI);
(*IcoFace19).SetFacePoints(IcoPointI, IcoPointL, IcoPointK);
(*IcoFace20).SetFacePoints(IcoPointK, IcoPointL, IcoPointD);

AddPoint(IcoPointA);
AddPoint(IcoPointB);
AddPoint(IcoPointC);
AddPoint(IcoPointD);
AddPoint(IcoPointE);
AddPoint(IcoPointF);
AddPoint(IcoPointG);
AddPoint(IcoPointH);
AddPoint(IcoPointI);
AddPoint(IcoPointJ);
AddPoint(IcoPointK);
AddPoint(IcoPointL);

//add faces to the face list
AddFace(IcoFace01);
AddFace(IcoFace02);
AddFace(IcoFace03);
AddFace(IcoFace04);
AddFace(IcoFace05);
AddFace(IcoFace06);
AddFace(IcoFace07);
AddFace(IcoFace08);
```

```
  AddFace(IcoFace09);
  AddFace(IcoFace10);
  AddFace(IcoFace11);
  AddFace(IcoFace12);
  AddFace(IcoFace13);
  AddFace(IcoFace14);
  AddFace(IcoFace15);
  AddFace(IcoFace16);
  AddFace(IcoFace17);
  AddFace(IcoFace18);
  AddFace(IcoFace19);
  AddFace(IcoFace20);

}


/*****************************************************************
 * Computes the height offset which is used as height scale     *
 * factor for a given point and refinement step.                *
 *****************************************************************/
float Planet::HeigtOffset(int step, Point* P)
{
  float l;
  //compute the basic factor using the position of the point
  Vector *VecP=(*P).GetPos();
  l=Rand.RandomContext(Seed, (*VecP)[0], (*VecP)[1], (*VecP)[2]);
  //reduce the value corresponding to the refinement step
  return (powf(2.0f,(float)-step)*l* HEIGHT_VARIANCE);
}


/*****************************************************************
 * Create a new point between the given points A and B,         *
 * including its height which is linear interplotated and than  *
 * displaced for a random amout.                                *
 *****************************************************************/
Point* Planet::CreatePoint(int ActualStep, Point* A, Point*B)
{
  //create a new vector and a new point
  float VecAB[3];
  Point* AB = new Point();

  //get their vectors of the parent points
```

```
  Vector *VecA=(*A).GetPos();
  Vector *VecB=(*B).GetPos();

  //compute the vector of the new point
  for(int i=0; i<3; i++)
  {
    VecAB[i]=(*VecA)[i]+(*VecB)[i];
  }

  //set the computet vectors and normalize it
  (*AB).SetPos(VecAB);
  (*AB).NormalizeVector();

  //compute the new HeightScaleFactors
  float hA, hB, h;
  hA=(*A).GetHeightScaleFactor();
  hB=(*B).GetHeightScaleFactor();
  h=(hA+hB)/2.0f+HeigtOffset(ActualStep, AB);
  (*AB).SetHeightScaleFactor(h);
  (*AB).DisplacePoint();

  //determine if its in polar zone
  bool AClimate=(*A).GetPolarZone();
  bool BClimate=(*B).GetPolarZone();
  float abY=(*AB).GetY();
  if(AClimate==BClimate)
    (*AB).SetPolarZone(AClimate);
  else if(abY>POLAR_CIRCLE || abY<-POLAR_CIRCLE)
    (*AB).SetPolarZone(TRUE);
  else
    (*AB).SetPolarZone(FALSE);

  //return the new point
  return AB;
}

/*************************************************************
 * Determines and returns, which type terrain a given triangle  *
 * (defined by three vertices) is. At the moment there are      *
 * three types of terrain. types are water (0), grass (1) and   *
 * ice (3).                                                     *
 *************************************************************/
char Planet::ComputeTerrain(Point* A, Point* B, Point* C)
```

```
{
  char terrain;

  //get the height for all vertices
  float ScaleFactorA=(*A).GetHeightScaleFactor();
  float ScaleFactorB=(*B).GetHeightScaleFactor();
  float ScaleFactorC=(*C).GetHeightScaleFactor();

  //if all points are high enough or in the polar zone, it's ice
  if(((ScaleFactorA>=1.040f)&&(ScaleFactorB>=1.040f)
                            &&(ScaleFactorC>=1.040f))
    || ((*A).GetPolarZone()==true && (*B).GetPolarZone()==true
                                  && (*C).GetPolarZone()==true))
  {
    terrain=TERRAIN_ICE;
  }
  //if the height of all points is zero, it's water
  else if((ScaleFactorA<=1.0f)&&(ScaleFactorB<=1.0f)
                              &&(ScaleFactorC<=1.0f))
  {
    terrain=TERRAIN_WATER;
  }
  //if its neither ice nor water, it must be grassland
  else
  {
    terrain=TERRAIN_GRASS;
  }
  return terrain;
}

/*****************************************************************
 * Refines each triangle of the faces list. Therefor a new      *
 * temporary list is create, each face is removed from the old  *
 * list, divided into four new faces and then deleted. The new  *
 * faces are added to the temporary list and at the end, the    *
 * temporary is set as new face list. This is the surface       *
 * refinement implementation used in the LOD part of the        *
 * algorithmn                                                   *
 *****************************************************************/
void Planet::RefineFaces(void)
{
  //temporary vector for faces
  std::list<Face*>* tmpFaceVector = new std::list<Face*>;
```

```
Face* tmpFace;
//for all faces
while(!(*Faces).empty())
{
  //get the last face and remove it from the vector
  tmpFace=(*Faces).back();
  (*Faces).pop_back();

  //create new faces
  Face* F1 = new Face();
  Face* F2 = new Face();
  Face* F3 = new Face();
  Face* F4 = new Face();

  //pointer for vertices of the old face
  Point* A;
  Point* B;
  Point* C;

  //get the vertices of the face
  A=(*tmpFace).GetFacePoint(0);
  B=(*tmpFace).GetFacePoint(1);
  C=(*tmpFace).GetFacePoint(2);

  //create the 3 new points
  Point* AB=CreatePoint(ActualStep, A, B);
  Point* AC=CreatePoint(ActualStep, A, C);
  Point* BC=CreatePoint(ActualStep, B, C);

  //and setup the new faces
  (*F1).SetFacePoints( A, AB, AC);
  (*F2).SetFacePoints( C, AC, BC);
  (*F3).SetFacePoints( B, BC, AB);
  (*F4).SetFacePoints(AB, BC, AC);

  //add the three new points to the point vector
  (*Points).push_back(AB);
  (*Points).push_back(AC);
  (*Points).push_back(BC);

  //add the four new faces to the new face vector
  (*tmpFaceVector).push_back(F1);
```

```
    (*tmpFaceVector).push_back(F2);
    (*tmpFaceVector).push_back(F3);
    (*tmpFaceVector).push_back(F4);

    //delete the old face
    delete tmpFace;
  }

  //set the temporary face vector as new face vector
  delete Faces;
  Faces=tmpFaceVector;
}

/*****************************************************************
 * Reduces the face in the face list depending on the position  *
 * of the virtual camera. In the first step the closest of the  *
 * vertices of the point list ist determined by calculating the *
 * difference vector. In the second step the faces list is      *
 * searched for all faces, that are using this vertex - these   *
 * are put in a temporary faces list and all vertices of these  *
 * faces are put into a temporary point list. these are the new *
 * face and point list of the planet - the old ones are deleted *
 *****************************************************************/
void Planet::ReduceFaces(Vector* Cam)
{
  double MinDiff=INFINITY;
  Point* Closest1=0;
  Point* Closest2=0;

  //temporary vectors for points and faces
  std::list<Point*>* tmpPointVector = new std::list<Point*>;
  std::list<Face*>* tmpFaceVector = new std::list<Face*>;

  //compute the closest point to the cam
  for(PointIterator=(*Points).begin();
      PointIterator != (*Points).end(); ++PointIterator)
  {
    Vector* p;
    Vector tmp;
    double Len=0.0f;
    p=(**PointIterator).GetPos();

    //compute the difference between the vectors
```

```
  for(int i=0; i<3; i++)
    tmp[i]=(*Cam)[i]-(*p)[i];

  //compute its length
  for(int i=0; i<3; i++)
    Len=Len+tmp[i]*tmp[i];
  Len=sqrt(Len);

  //choose the as new closest point if its difference is lower
  if(Len<MinDiff)
  {
    MinDiff=Len;
    Closest1=(*PointIterator);
    Closest2=0;
  }
  //if they have the same length
  else if(Len==MinDiff)
  {
    //test if they are identic
    bool identic=TRUE;
    Vector* ClosestVec=(*Closest1).GetPos();
    Vector* PointVec=(**PointIterator).GetPos();
    for(int i=0; i<3; i++)
    {
      if((*ClosestVec)[i]!=(*PointVec)[i])
      {
        identic=FALSE;
        break;
      }
    }
    //if they are identic, remember this as closest point 2
    if(identic)
    {
      Closest2=(*PointIterator);
    }
  }
}

//find the faces connected to the closest point
Face* tmpFace;
while(!(*Faces).empty())
{
  //get the last face and remove it from the vector
```

```
  tmpFace=(*Faces).back();
  (*Faces).pop_back();

  //if tmpFace has the closest point one
  //add it to the new face list
  int p1=(*tmpFace).HasFacePoint(Closest1);
  if(p1>=0)
    (*tmpFaceVector).push_back(tmpFace);
  //else if closest point 2 exist
  else if(Closest2!=0)
  {
    //if tmpFace has the closest point two,
    int p2=(*tmpFace).HasFacePoint(Closest2);
    if(p2>=0)
    {
      //set this point to the first closest point
      (*tmpFace).SetFacePoint(p2, Closest1);
      //and add it to the new face list
      (*tmpFaceVector).push_back(tmpFace);
    }
    //if this face has neither closest point one or two
    else
      //it can be deleted
      delete tmpFace;
  }
}

//set the new face vector to the temporary face vector
delete Faces;
Faces=tmpFaceVector;

Point* tmpPoint;
while(!(*Points).empty())
{
  //get the last point and remove it from the vector
  tmpPoint=(*Points).back();
  (*Points).pop_back();
  //test all faces
  bool newPoint=FALSE;
  for(FaceIterator = (*Faces).begin();
      FaceIterator != (*Faces).end(); ++FaceIterator)
  {
    //if one of the faces has the this point
```

```
      if((**FaceIterator).HasFacePoint(tmpPoint)>=0)
      {
        newPoint=TRUE;
        //test, if its already in the tempary point vector
        for(PointIterator=(*tmpPointVector).begin();
            PointIterator != (*tmpPointVector).end();
            ++PointIterator)
        {
          if((*PointIterator)==tmpPoint)
          {
            newPoint=FALSE;
            break;
          }
        }
      }

    //if it is a new point, add it to the temporary point vector
    if (newPoint)
      (*tmpPointVector).push_back(tmpPoint);
    //else delete it
    else
      delete tmpPoint;
  }
  delete Points;
  Points=tmpPointVector;

}

/*****************************************************************
 * Performs some sort of level of detail by iterativly calling  *
 * of RefineFaces() and ReduceFaces(). The number of the calls  *
 * is determined by the number of the MaxSteps and the DETAIL   *
 * constant.                                                     *
 *****************************************************************/
void Planet::LOD(Vector* Cam)
{
  //reduce the faces
  ReduceFaces(Cam);
  //refine and reduce faces until there are only as much steps
  //left as specified by the DETAIL constant
  for(int i=0; i<MaxSteps-DETAIL; i++)
  {
```

```
    ActualStep++;
    RefineFaces();
    ReduceFaces(Cam);
  }
}

/*****************************************************************
 * Generates the planet from a base solid. The parameters are   *
 * the position of the virtual camera and the number of steps.  *
 * The camera position is used to find the part of the planet,  *
 * that is to be rendered and the number of the steps is used,  *
 * to
 *****************************************************************/
void Planet::GeneratePlanet(Vector* Cam, int steps)
{
  Points = new std::list<Point*>;
  Faces = new std::list<Face*>;

  MaxSteps=steps;
  ActualStep=0;

  //defines the base solid for the planet (here a icosahedron)
  Solid();

  //if there are more steps left than the desired detail
  if (MaxSteps>DETAIL)
    //start the LOD part of the algoritmn
    LOD(Cam);

  //do the recursive part of the algoritmn for all faces left
  for(FaceIterator = (*Faces).begin();
      FaceIterator != (*Faces).end(); ++FaceIterator)
  {
    //get the vertices for each face
    Point* A=(**FaceIterator).GetFacePoint(0);
    Point* B=(**FaceIterator).GetFacePoint(1);
    Point* C=(**FaceIterator).GetFacePoint(2);

    if(MaxSteps>DETAIL)
      Recursive(A, B, C, ActualStep+1, DETAIL);
    else
      Recursive(A, B, C, ActualStep+1, MaxSteps);
  }
```

```
  //clean up, delete the faces
  Face* tmpFace;
  while(!(*Faces).empty())
  {
    //get the last face, remove it from the vector and delete it
    tmpFace=(*Faces).back();
    (*Faces).pop_back();
    delete tmpFace;
  }
  delete Faces;

  //clean up, delete the points
  Point* tmpPoint;
  while(!(*Points).empty())
  {
    //get the last point, remove it from the vector and delete it
    tmpPoint=(*Points).back();
    (*Points).pop_back();
    delete tmpPoint;
  }
  delete Points;

}

/*************************************************************
 * The recursive part of the algorithmn. In this part a face is *
 * divided recursivly. The face is defined by three vertices A, *
 * B and C. actualStep is used to compute the correct height of *
 * created vertices and RemainigSteps determines the end of the *
 * recursion.                                                   *
 *************************************************************/
void Planet::Recursive(Point* A, Point* B, Point* C,
                       int actualStep, int RemainigSteps)
{

  //if there a more steps left
  if(RemainigSteps>0)
  {
    //create the new points from the vertices of the old face
    Point* AB=CreatePoint(actualStep, A, B);
    Point* AC=CreatePoint(actualStep, A, C);
    Point* BC=CreatePoint(actualStep, B, C);
```

```
    //add the new points to the point list
    AddPoint(AB);
    AddPoint(AC);
    AddPoint(BC);

    //divide recursive the new calculated faces
    Recursive( A, AB, AC, actualStep+1, RemainigSteps-1);
    Recursive( C, AC, BC, actualStep+1, RemainigSteps-1);
    Recursive( B, BC, AB, actualStep+1, RemainigSteps-1);
    Recursive(AB, BC, AC, actualStep+1, RemainigSteps-1);
  }
  //if there are no more steps left
  else
  {
    //get the postion of the displaced points
    //this is the face to be rendered
    Vector *VecA=(*A).GetDisplacedPos();
    Vector *VecB=(*B).GetDisplacedPos();
    Vector *VecC=(*C).GetDisplacedPos();

    //compute the type of the area to get the right colors
    char type=ComputeTerrain(A, B,C);
    //call the GraphicCore to draw the face
    GC.DrawTriangle(*VecA, *VecB, *VecC, type);
  }

}

/*************************************************************
 * Simply adds a new point to the point list. nothing special.  *
 *************************************************************/
void Planet::AddPoint(Point* P)
{
  (*Points).push_back(P);
}

/*************************************************************
 * Simply adds a new face to the faces list. nothing special.   *
 *************************************************************/
void Planet::AddFace(Face* F)
{
  (*Faces).push_back(F);
```

```
}

/*****************************************************************
 * The destructor of the class. nothing special in there.       *
 *****************************************************************/
Planet::~Planet(void)
{
}
```

B.3 Face.h

```
#ifndef FACE_H
#define FACE_H

#include "Point.h"

class Face
{
private:
  //face consists of three pointers to points
  Point* A;
  Point* B;
  Point* C;

public:
  //constructor
  Face(void);
  //set one of the three vertices of the face
  void SetFacePoint(int, Point*);
  //set all three vertices of the face
  void SetFacePoints(Point*, Point*, Point*);
  //get one of the vertices
  Point* GetFacePoint(int);
  //check if a given vertex is part of the face
  int HasFacePoint(Point*);
  //destructor
  ~Face(void);
};

#endif
```

B.4 Face.cpp

```
#include "Face.h"
/*****************************************************************
 * Initializes the vertices of the face with null pointers.      *
 *****************************************************************/
Face::Face(void)
{
  A=0;
  B=0;
  C=0;
}

/*****************************************************************
 * Sets the point P as face point i.                             *
 *****************************************************************/
void Face::SetFacePoint(int i, Point* P)
{
  switch (i)
  {
    case 0: A=P; break;
    case 2: B=P; break;
    case 1: C=P; break;
  }

}

/*****************************************************************
 * Sets all three face points according to the three parameter. *
 *****************************************************************/
void Face::SetFacePoints(Point* a, Point* b, Point* c)
{
  A=a;
  B=b;
  C=c;
}

/*****************************************************************
 * Returns a pointer to the face vertex specified by the        *
 * parameter i, if i is between 0 and 2.                         *
 *****************************************************************/
Point* Face::GetFacePoint(int i)
{
```

```
  switch (i)
  {
    case 0: return A;
    case 1: return B;
    case 2: return C;
    default: return 0;
  }
}

/*****************************************************************
 * This method checks, if the point, given by the parameter, is *
 * one of the vertices of the face. If so, it returns, which    *
 * one it is, if not, it returns -1.                             *
 *****************************************************************/
int Face::HasFacePoint(Point* P)
{
  if (A==P) return 0;
  else if (B==P) return 1;
  else if (C==P) return 2;
  else return -1;
}

/*****************************************************************
 * The destructor of the class. nothing special in there.       *
 *****************************************************************/
Face::~Face(void)
{
}
```

B.5 Point.h

```
#ifndef POINT_H
#define POINT_H

#include "TypeDefs.h"
#include <cmath>


class Point
{
private:
```

```
  //vector of the position of the point on the surface of
  //the sphere
  Vector Pos;
  //vector of the position of the point on the surface of
  //the planet
  Vector DisplacedPos;
  //the height of the point above the surface of the sphere
  float HeightScaleFactor;
  //true if the point is part of the polar zoone
  bool Polar;

public:
  //constructor
  Point(void);
  //set the position of the point to the given vector
  void SetPos(Vector);
  //set the position of the point to the given coordinates
  void SetPos(float, float, float);
  //returns a pointer to the position of the point
  Vector* GetPos(void);
  //returns y-coordinate of the position of the point
  float GetY();
  //sets the height scale factor to the given value
  void SetHeightScaleFactor(float);
  //returns the height scale factor
  float GetHeightScaleFactor(void);
  //set the polar zone property of the point to the given value
  void SetPolarZone(bool c);
  //returns true if the point is part of one polar zone
  bool GetPolarZone(void);
  //returns a pointer to the DisplacedPos vector, needed for
  //the rendering of the planet
  Vector* GetDisplacedPos(void);
  //computes the position of the point according to
  //height scale factor
  void DisplacePoint(void);
  //normalizes the Pos vector of the point
  void NormalizeVector(void);
  //destructor
  ~Point(void);
};

#endif
```

B.6 Point.cpp

```
#include "Point.h"

/*****************************************************************
 * The constructor of the class. Initializes the values for the *
 * vectors Pos and DisplacedPos, the height scale factor and    *
 * the polar zone property.                                      *
 *****************************************************************/
Point::Point(void)
{
  Pos[0]=0.0f;
  Pos[1]=0.0f;
  Pos[2]=0.0f;

  DisplacedPos[0]=0.0f;
  DisplacedPos[1]=0.0f;
  DisplacedPos[2]=0.0f;

  HeightScaleFactor=1.0f;
  Polar=false;
}

/*****************************************************************
 * Sets the Pos vector (the position of the point on the sphere *
 * surface) according to the vector given by the parameter.     *
 *****************************************************************/
void Point::SetPos(Vector pos)
{
  for(int i=0; i<3; i++) Pos[i]=pos[i];
}

/*****************************************************************
 * Sets the Pos vector (the position of the point on the sphere *
 * surface) according to the parameters x, y and z.             *
 *****************************************************************/
void Point::SetPos(float x, float y, float z)
{
  Pos[0]=x;
  Pos[1]=y;
  Pos[2]=z;
}
```

```
/*****************************************************************
 * Returns the position vector of a point as reference.          *
 *****************************************************************/
Vector* Point::GetPos()
{
  return &Pos;
}

/*****************************************************************
 * Returns the y-coordinate of the vertex. this is used to       *
 * determine, if a point is north or south of the polar circle.  *
 *****************************************************************/
float Point::GetY()
{
  return Pos[1];
}

/*****************************************************************
 * Sets the height scale factor (the height of the point on the  *
 * surface of the planet) to a the value given by the            *
 * parameter.                                                    *
 *****************************************************************/
void Point::SetHeightScaleFactor(float hsf)
{
  HeightScaleFactor=hsf;
}

/*****************************************************************
 * Returns the height scale factor (the height of the point      *
 * the sphere surface) as floating point value.                  *
 *****************************************************************/
float Point::GetHeightScaleFactor(void)
{
  return HeightScaleFactor;
}

/*****************************************************************
 * Sets the polar zone property of the point according to the    *
 * parameter p.                                                  *
 *****************************************************************/
void Point::SetPolarZone(bool p)
{
  Polar=p;
```

```
}

/*****************************************************************
 * Returns true, if the point is part of the polar zone, else   *
 * false.                                                        *
 *****************************************************************/
bool Point::GetPolarZone(void)
{
  return Polar;
}

/*****************************************************************
 * Scales the vector Pos (position on the sphere) with the      *
 * height scale factor and saves the result in DisplacePos.     *
 * This is then the position of the point to be rendered .      *
 *****************************************************************/
void Point::DisplacePoint(void)
{
  float hsf;
  //get the scale factor and if it is below 1, set it to 1
  if(HeightScaleFactor<1.0f)
    hsf=1.0f;
  //else it is the HeightScaleFactor
  else
    hsf=HeightScaleFactor;

  //displace the point by mutliplying its components with the
  //scale factor
  DisplacedPos[0]=hsf * Pos[0];
  DisplacedPos[1]=hsf * Pos[1];
  DisplacedPos[2]=hsf * Pos[2];
}

/*****************************************************************
 * Return the vector with the displaced position. this is       *
 * needed for the rendering of the planet.                      *
 *****************************************************************/
Vector* Point::GetDisplacedPos()
{
  return &DisplacedPos;
}

/*****************************************************************
```

```
 * This method normalizes the vector of the position of the     *
 * point, so all points have the same distance from the origin. *
 ****************************************************************/
void Point::NormalizeVector(void)
{
  //normalize a vector
  float d = 0.0f;

  //get it's length
  for(int i=0; i<3; i++)
    d = d + Pos[i] * Pos[i];
  d = sqrtf(d);

  if(d>0.0f) //avoid division by zero
    for(int i=0; i<3; i++)
      Pos[i]=Pos[i]/d;

}

/****************************************************************
 * The destructor of the class. nothing special in there.       *
 ****************************************************************/
Point::~Point(void)
{
}
```

B.7 Random.h

```
#include <cmath>

class Random
{

public:
  //constructor - nothing in there
  Random(void);
  //takes an int an three double values and results an
  //pseudo random float depending on the parameters
  float RandomContext(int, double, double, double);
  //constructor - nothing in there
  ~Random(void);
};
```

B.8 Random.cpp

```
#include "Random.h"

/*************************************************************
 * Nothing in there.                                         *
 *************************************************************/
Random::Random(void)
{
}

/*************************************************************
 * Creates an pseudo random value, depending on its parameters. *
 * The first parameter is an integer - the seed of the planet.  *
 * The other parameters are double values - these are the three *
 * coordinates of a vertex. The double values are converted     *
 * into integer then an pseudo random value is computed with a  *
 * modified version of Hugo Elias formula.                      *
 *************************************************************/
float Random::RandomContext(int seed,
                            double x, double y, double z )
{
  //convert the floating point values into integer
  unsigned long i1=(unsigned long) ceil(x*100000.0f);
  unsigned long i2=(unsigned long) ceil(y*100000.0f);
  unsigned long i3=(unsigned long) ceil(z*100000.0f);

  //adjust Hugo Elias formula
  signed long n = seed + i1 * 57 + i2 * 127 + i3 * 307;

  //use Hugo Elias formula
  n = (n<<13) ^ n;
  float r = (float)( 1.0 - ( (n * (n * n * 15731 + 789221)
                  + 1376312589) & 0x7fffffff ) / 1073741824.0);
  return r;
}

/*************************************************************
 * Nothing in there.                                         *
 *************************************************************/
Random::~Random(void)
{
}
```

Literaturverzeichnis

[1] Benoît B. Mandelbrot: *The Fractal Geometry of Nature*. New York, W. H. Freeman and Co., 1982.

[2] F. Kenton Musgrave: *Methods for Realistic Landscape Imaging*. Yale University, 1993.

[3] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley: *Texturing and modeling: a procedural approach*. Academic Press, Chestnut Hill, Massachusetts, second edition, 1998.

[4] Terragen: `http://www.planetside.co.uk/terragen/`

[5] Bryce: `http://bryce.daz3d.com/`

[6] Paul Bourke: `http://astronomy.swin.edu.au/~pbourke/modelling/sphere/`

[7] Stefan Rahn: *Out-of-Core Level of Detail*. Studienarbeit, Universität Rostock, Fachbereich Informatik, 2005. `http://www.icg.informatik.uni-rostock.de/archiv/Studenten/Studienarbeiten/2005/Stefan_Rahn/docs/SA_SRahn2005.pdf/`

[8] James H. Clark: *Hierarchical Geometric Models for Visible Surface Algorithms*. Communications of the ACM. Volume 19, Issue 10, 1976.

[9] The Stanford 3D Scanning Repository: `http://graphics.stanford.edu/data/3Dscanrep/`

[10] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, Gregory A. Turner: *Real-time continuous level of detail rendering*

of height fields. Computer Graphics (SIGGRAPH 96 Proceedings), p.109-118, 1996.

[11] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein: *ROAMing Terrain: Real-time Optimally Adapting Meshes*. IEEE Visualization '97 Proceedings, 1997.

[12] Stefan Röttger, Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel: *Real-Time Generation of Continuous Levels of Detail for Height Fields*. Proceedings of the WSCG '98, 1998. `http://www9.informatik.uni-erlangen.de/Persons/Roettger/papers/TERRAIN.PDF`

[13] Hugo Elias: `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`

[14] Alain Fournier, Don Fussell, Loren Carpenter: *Computer Rendering of Stochastic Models*. Communications of the ACM, 1982.

[15] Ken Perlin: *Improving Noise*. Computer Graphics, Vol. 35, No. 3, 2002.

[15] Jochen Winzen:*Interaktive Visualisierung eines Planetensystems*. Studienarbeit, 2003.

[17] Ken Perlin: `http://mrl.nyu.edu/~perlin/experiments/demox/Planet.html`

[20] OpenRT: `http://www.openrt.de`

[21] SaarCOR: `http://www.saarcor.de`

[22] SpeedTree: `www.speedtree.com`