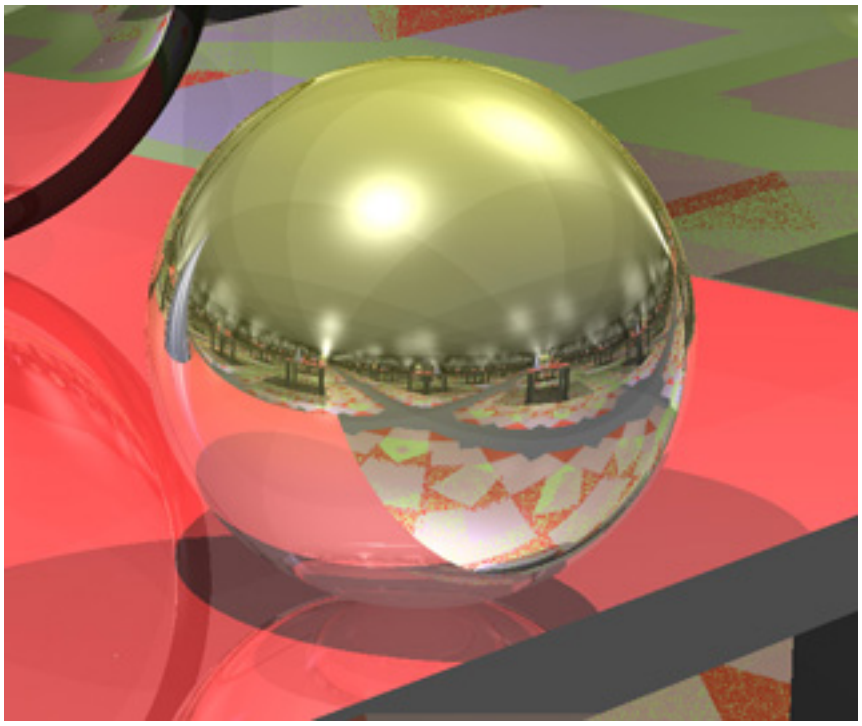
Computergraphik

Proseminar - Michael J. Winckler



Sommersemester 2006

Ausarbeitungen, Stand 05. Juni 2007

Inhaltsverzeichnis

Einleitung

Das Proseminar *Computergraphik* fand im Sommersemester 2006 in der Arbeitsgruppe *Visualisierung und Numerische Geometrie* an der Universität Heidelberg statt.

Die hier zusammengestellten Proseminarausarbeitungen sind in der Abfolge der Vorträge aufgeführt.

Heidelberg, Juni 2007

Michael Winckler[1]

Teilnehmerliste und Themen

Homogene Koordinaten – Ansgar Borchardt
Beleuchtung und Lichtmodelle – Benjamin Rommel
Direct Remndering mit OpenGL – Alexander Ortenburger
Texturen und Texture Mapping – Maik Häsner
Bezier-Kurven – Jens Fangerau
Noise und Turbulence – Thomas Zessin Partikelsysteme – Lorenz Köhl
Autonome Agenten und Steering – Hendrik Schmidt
VRML-Modelierung – Sharon Friedrich
Path-Findung und A-Stern – Roger Gaczkowski
Delaunay-Triangulierung – Sebastian Bechtiold
Terrain-Visualisierung – Daniel Jungblut

[1] IWR, Universiät Heidelberg, Im Neuenheimer Feld 368, 69120 Heidelberg, Michael.Winckler@iwr.uni-heidelberg.de

Kapitel 1

Homogene Koordinaten in der Computergrafik

1.1 Einleitung

Zur Darstellung von Objekten und ganzen Welten in Computern wird ein Koordinatensystem verwendet. Intuitiv liegt es nahe hierfür die bekannten euklidischen Koordinaten zu verwenden. Es hat sich aber früh herausgestellt, dass für eine elegantere Behandlung eine anderes Koordinatensystem, die *homogenen Koordinaten* geeigneter sind.

In dieser Arbeit sollen die homogenen Koordinaten vorgestellt werden und gezeigt werden, wie man mit ihnen arbeitet. Wir betrachten zunächst die Darstellung von einfachen Objekten, wie man diese verändern kann und letztlich auf dem Monitor dargestellt werden können. Letztendlich lassen sich zwar alle Rechnungen auch mit euklidischen Koordinaten durchführen, aber diese sind in der Behandlung aufwendiger.

1.2 Homogene Koordinaten

Wir wollen die homogenen Koordinaten für den $\mathbb{R}^n$ einführen. Hierzu definieren wir zunächst eine Relation auf dem $\mathbb{R}^{n+1}$:

$$a \sim b : \Longleftrightarrow \text{ es gibt } \lambda \neq 0 \text{ mit } a = \lambda b$$

a und b stehen also in Relation zueinander genau dann wenn a ein Vielfaches von b ist. Man kann zeigen, dass $\sim$ eine *Äquivalenzrelation* ist.

Die *Quotientenmenge* $P^n := \mathbb{R}^{n+1}/\sim$ bildet mit der Addition

$$[a] + [b] := [a + b]$$

und der Skalarmultiplikation

$$\lambda[a] := [\lambda a]$$

einen $\mathbb{R}$-Vektorraum. Wir bezeichnen P^n als *projektive Ebene*. Die Addition und Skalarmultiplikation verhalten sich anders als im R^{n+1}: Die Summe $A + B$ zweier Punkte $A, B \in P^n$ ergibt den Mittelpunkt der Strecke $\overline{AB}$; die skalare Multiplikation mit einer Zahl $\neq 0$ liefert einen anderen Repräsentanten der selben Äquivalenzklasse, ändert also nichts am Punkt selbst.

Wir identifizieren nun $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ mit den *homogenen Koordinaten* $x' = (x_1, \ldots, x_n, 1) \in P^n$ und umgekehrt.

Wir bezeichnen diese $x' = (x_1, \ldots, x_{n+1}) \in P^n$ mit $x_{n+1} \neq 0$ als *Punkte*. Die Differenzen zweier Punkte haben die Form $v' = (v_1, \ldots, v_n, 0)$. Wir bezeichnen alle solche v' als *Vektoren*.

Auf der projektiven Ebene muss also zwischen Punkte und Vektoren unterschieden werden. Es handelt sich um einen affinen Vektorraum.

Um uns später die Rechnungen zu vereinfachen führen wir noch die normalisierte Darstellung von Punkten ein: Zu jedem Punkt $x' = (x_1, \ldots, x_{n+1}) \in P^n$ mit $x_{n+1} \neq 0$ gibt es einen Repräsentanten der Form $(x_1, \ldots, x_n, 1)$.[1] Wir bezeichnen einen solchen Repräsentanten als *normalisiert*. Im Folgenden werde ich immer von normalisierten homogenen Koordinaten ausgehen.

1.3 Objekte in homogenen Koordinaten

Einzelne Punkte können wir leicht in homogenen Koordinaten beschreiben, indem wir $p = (p_1, \ldots, p_n) \in \mathbb{R}^n$ mit dem Punkt $p' = (p_1, \ldots, p_n, 1) \in P^n$ identifizieren. Im Folgenden wird dargestellt, wie einige weitere einfache Objekte, wie Geraden, Strecken und Ebenen, in homogenen Koordinaten dargestellt werden können.

Mit komplexeren Elementen wir Polygonen, Kugeln usw. werde ich mich an dieser Stelle nicht beschäftigen.

1.3.1 Geraden und Strecken

Im $\mathbb{R}^n$ läßt sich eine Gerade mit Stützpunkt u und Richtungsvektor v bekanntlich darstellen als

$$G = \{a + \lambda v | \lambda \in \mathbb{R}\}$$

Also hat jedes $x \in G$ die Form $x = a + \lambda u$. Übertragen wir diesen Punkt in homogene Koordinaten, so erhalten wir $x' = (a_1 + \lambda u_1, \ldots, a_n + \lambda u_n, 1)$, also

$$G' = \{(a + \lambda u, 1) | \lambda \in \mathbb{R}\}$$

[1] Um diesen Repräsentanten zu erhalten multipliziere x' mit x_{n+1}^{-1}

Oder, falls wir den Stützpunkt als Punkt und den Richtungsvektor als Vektor schreiben wollen,

$$G' = \{(a, 1) + \lambda(u, 0) | \lambda \in \mathbb{R}\}$$

1.3.2 Ebenen

Ebenen werden wie im $\mathbb{R}^n$ von zwei Geraden aufgespannt:

$$E = \{a + \lambda u + \mu v | \lambda, \mu \in \mathbb{R}\}$$

wobei a ein Stützpunkt ist und u, v Richtungsvektoren sind.

Im dreidimensionalen Fall gibt es zusätzlich die Normalenform:

$$E = \{< n, p - a >= 0 | p \in P^3\}$$

wobei $n \perp E$ ein senkrecht auf der Ebene stehender Vektor ist (Normalenvektor) und $<, >$ das Skalarprodukt bezeichnet.

Man sieht, dass Objekte in homogenen Koordinaten ähnlich behandelt werden können wie im euklidischen $\mathbb{R}^n$, es sind nur kleinere Anpassungen nötig.

1.4 Affine Transformationen

Affine Transformationen sind Abbildungen, bei der parallele Geraden parallel bleiben. Allgemein läßt sich jede solche Abbildung in euklidischen Koordinaten schreiben als

$$P' = M \cdot P + t$$

wobei M eine $n \times n$-Matrix und $t \in \mathbb{R}^n$ ein Vektor ist. Im Spezialfall $t = 0$ spricht man von linearen Abbildungen.

In homogenen Koordinaten lassen sich affine Transformationen etwas kompater schreiben als

$$(P', 1) = M' \cdot (P, 1)$$

wobei M' eine $(n + 1) \times (n + 1)$-Matrix ist.

Affine Transformationen werden in der Computergrafik an vielen Stellen benötigt, z.B. um Objekte zu skalieren oder in einer Welt zu positioneren. Im Folgenden werden die wichtigesten Transformationen vorgestellt. Auch wichtig ist die Möglichkeit mehrere Transformationen zu verknüpfen und so aus einfachen Grundtransformationen komplexere aufzubauen.

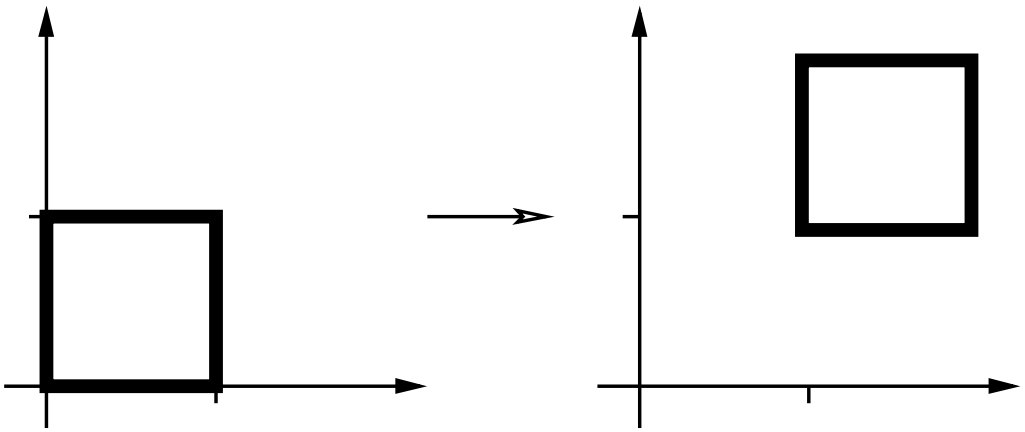


Abbildung 1.1: Translation um den Vektor (1, 1)

1.4.1 Translation

Eine Translation ist eine Verschiebung. In euklidischen Koordinaten läßt sie sich schreiben als

$$P' = P + t$$

wobei P' der verschobene Punkt, P der alte Punkt und t der Verschiebungsvektor ist.

Übertragen auf (normalisierte) homogene Koordinaten läßt sich die Translation als Matrixoperation schreiben:

$$\begin{pmatrix} P' \\ 1 \end{pmatrix} = M \cdot \begin{pmatrix} P \\ 1 \end{pmatrix}$$

wobei

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ t^t & 1 \end{bmatrix}$$

In der Abb. 1.1 ist eine Verschiebung eines Quadrats um den Vektor $t = (1, 1)$ auf der zweidimensionalen Ebene dargestellt. In der Matrixschreibweise würde dies bedeuten

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

1.4.2 Skalierung

Die Skalierung eines Objekts ist eine Verzerrung. Im einfachsten Fall, der *uniformen Skalierung* wird in alle Richtungen um den gleichen Faktor gestreckt. Das Ergebnis ist eine Vergrößerung (bzw. Verkleinerung) des Objekts.

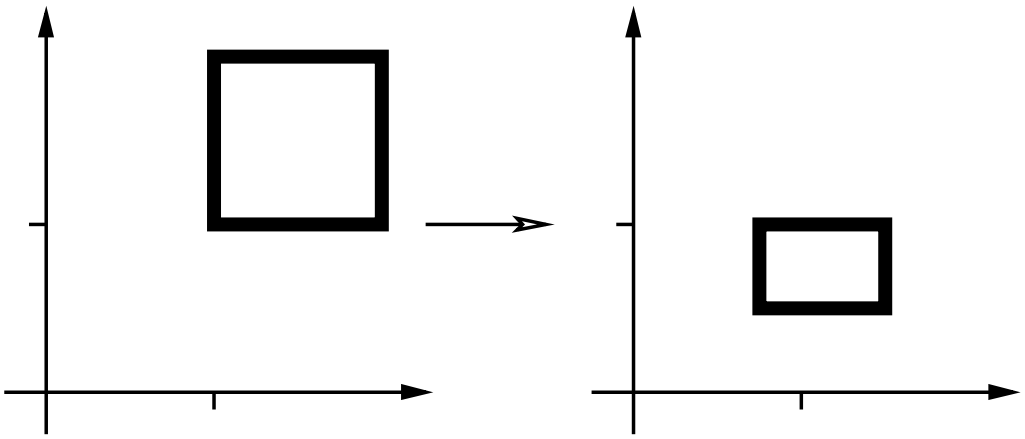


Abbildung 1.2: Skalierung um 0.75 in x-Richtung, 0.5 in y-Richtung

Allgemein wird eine Skalierung von drei Faktoren s_x, s_y und s_z bestimmt, die die Verzerrung in Richtung der jeweiligen Koordinatenachse angeben. Faktoren > 1 bewirken eine Streckung, Faktoren < 1 eine Stauchung.

In homogenen Koordinaten läßt sich die Skalierung wie folgt als Matrixoperation schreiben:

$$P' = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot P$$

Die Abbildung 1.2 zeigt eine Stauchung in Richtung beider Koordinatenachsen. Da sich das Objekt hierbei nicht im Koordinatenursprung befindet ändert sich auch der Abstand von diesem. Möchte man dies nicht, muss man das Objekt vor der Skalierung erst mittels einer Translation passend verschieben.

1.4.3 Rotation

Wir betrachten zunächst eine Drehung um den Winkel φ um die z-Achse. Offensichtlich ändert sich hierbei die z-Koordinate nicht, also müssen wir nur noch die Drehung in der xy-Ebene betrachten:

Wir können die Drehung auch als Basiswechsel auffassen. Aus der linearen Algebra ist bekannt, wie wir dies in Matrixgestalt schreiben können. Mit Hilfe von Abbildung 1.3 erhalten wir die Matrix

$$M_{z,\varphi} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

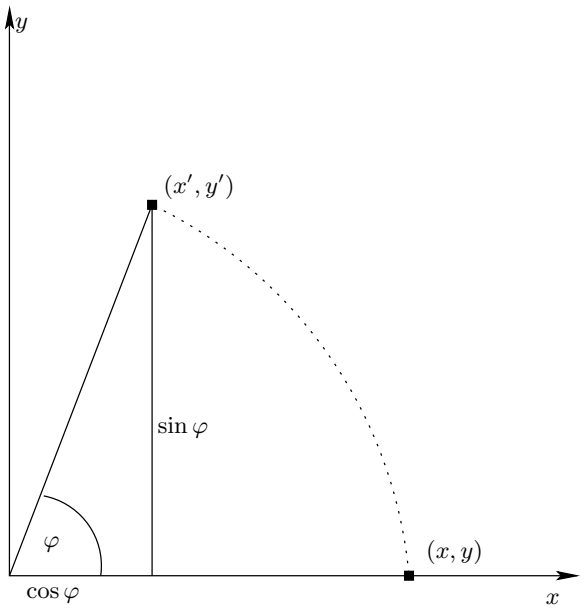


Abbildung 1.3: Rotation in der Ebene

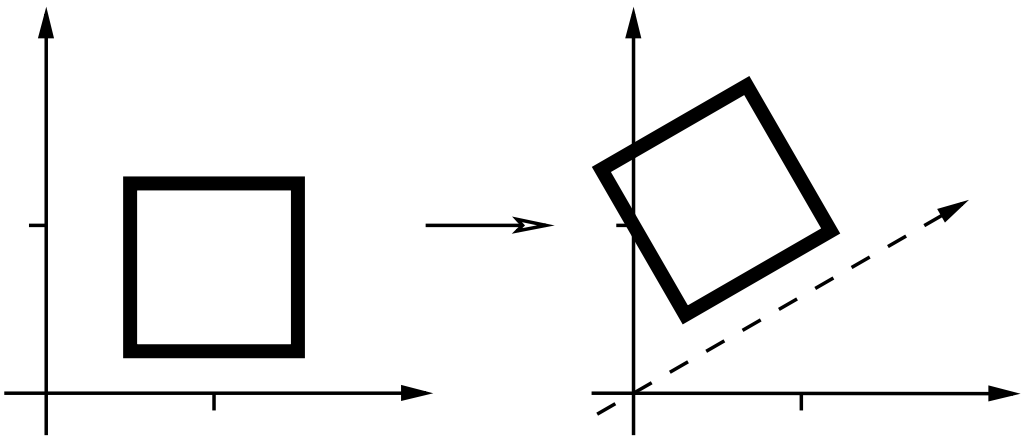


Abbildung 1.4: Rotation um 30 Grad um den Ursprung

Analog erhalten wir für die Rotation um die x- bzw. y-Achse die Matrizen

$$M_{x,\varphi} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M_{y,\varphi} = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Rotation um beliebige Achsen kann man durch Verkettung der Rotationen um die einzelnen Koordinatenachsen erhalten. Alternativ kann man den Basiswechsel zwischen altem Koordinatensystem und dem neuen, gerehten Koordinatensystem betrachten und so eine Transformationsmatrix erhalten.

1.4.4 Verkettung mehrerer Transformationen

Bis jetzt können wir erst sehr einfache Transformationen durchführen: Translationen, Skalierungen, Scherungen und Rotationen um den Ursprung. Aber aus diesen lassen sich einfach durch Verkettung komplexe Transformationen, wie z.B. die Rotation um eine beliebige Achse, zusammensetzen.

Wollen wir zwei Transformationen (in Form der Matrizen T_1 und T_2) hintereinander auf einen Punkt P anwenden, so wenden wir einfach die Transformation T_2 auf das Ergebnis P' der ersten Transformation an:

$$P'' = T_2 P' = T_2(T_1 P)$$

Aus der linearen Algebra ist bekannt, das für die Matrizenmultiplikation das Assoziativgesetz gilt. Mit der "zusammengefassten" Transformation $T_{12} = T_2 T_1$ (die Reihenfolge ist hier wichtig!) erhalten wir

$$P'' = T_{12} P.$$

Dies läßt sich leicht auf beliebig lange Ketten von Transformationen erweitern. Da oft dieselbe Transformation auf viele Punkte auf einmal angewendet wird, spart dies viele Rechenschritte und somit wird weniger Hardware benötigt.

Als Beispiel betrachten wir eine Translation um den Vektor $(1, 1, 1)$ und anschließender (uniformer) Skalierung um den Faktor 2:

$$M = \underbrace{\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Skalierung}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Translation}} = \begin{bmatrix} 2 & 0 & 0 & -2 \\ 0 & 2 & 0 & -2 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

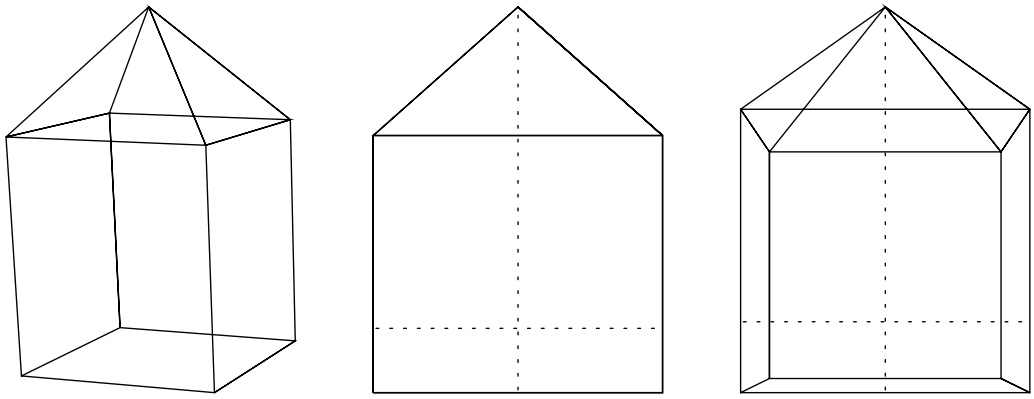


Abbildung 1.5: Ein Haus dargestellt mit Parallelprojektion (Mitte) und Zentralprojektion (Links, Rechts)

1.5 Projektionen

Mit Hilfe einer Projektion wird in der Computergrafik eine dreidimensionale Beschreibung einer Welt eine Ansicht dieser auf eine zweidimensionale Fläche abgebildet.

Nach der Projektion hat man eine zweidimensionale Abbildung der Welt, ganz fertig ist diese jedoch nicht: Es dürfen nur Objekte dargestellt werden, die nicht von anderen Objekten verborgen sind. Für die Darstellung auf einem Monitor muss das Bild auch noch gerastert werden, d.h. in einzelne Bildpunkte aufgeteilt werden. Mit diesen Schritten werden wir uns aber an dieser Stelle nicht beschäftigen.

1.5.1 Parallelprojektion

Wir gehen im Folgenden davon aus, dass wir vom Koordinatenursprung in Richtung der z-Achse schauen wollen. Dies kann mittels Translation und Rotation erreicht werden.

Die Parallelprojektion ist nun sehr einfach: Wir projezieren alles auf die xy-Ebene indem wir einfach die z-Koordinate auf 0 setzen. In Matrixschreibweise bedeutet dies:

$$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot P.$$

Natürlich muss hierbei eigentlich nicht gerechnet werden, sondern ein Programm kann einfach so tun, als ob die z-Koordinate 0 ist.

Die Parallelprojektion ist nicht perspektivisch: Weiter hinten stehende Objekte wirken nicht kleiner (klar, da die hierfür relevante z-Koordinate keine Beachtung findet). Dafür erhält sie jedoch Längen auf der xy-Achse.

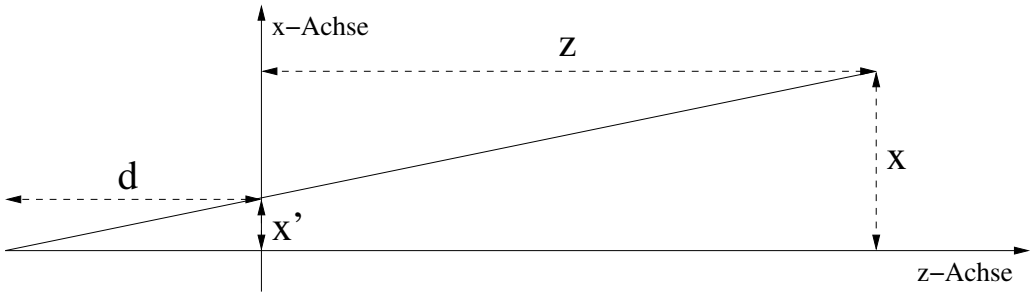


Abbildung 1.6: Herleitung der Formel für die Zentralprojektion

Aus diesem Grund ist die Parallelprojektion wichtig für computergestütztes Design, aber nicht für die realistische Darstellung von Welten in z.B. Spielen oder Filmen.

1.5.2 Zentralprojektion

Wir gehen wieder davon aus, dass wir vom Punkt $(0, 0, -d)$ auf die xy-Ebene projezieren. d wird als *Augabstand* bezeichnet.

Wir betrachten einen Punkt $P = (x, y, z, 1) \in P^n$ Aus Abbildung 1.6 kann man mittels Strahlensatz ablesen, dass für die Koordinate x' der Projektion gilt $x' = \frac{d}{z+d} \cdot x$. Analog für die y-Koordinate. Außerdem muss offensichtlich $z = 0$ gelten.

Als Matrixoperation können wir die Zentralprojektion schreiben als

$$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \cdot P.$$

Dies ist nur möglich, da P' als Äquivalenzklasse mehrere Darstellungmöglichkeiten hat. In euklidischen Koordinaten lässt sich die Zentralprojektion *nicht* als Matrixoperation schreiben!

Wie man in Abbildung 1.5 sieht, ist die Zentralprojektion perspektivisch, d.h. weiter entfernte Dinge werden kleiner dargestellt. Bei Verwendung der Zentralprojektion schneiden sich ursprünglich parallele Linien, die nicht parallel zur Projektionsebene sind, in einem Fluchtpunkt, den man durch Projektion des Richtungsvektors erhalten kann. Bei der Zentralprojektion bleiben jedoch Längen nicht erhalten und Winkel auch nur parallel zur Projektionsebene.

Die Zentralprojektion wird z.B. in 3D-Computerspielen verwendet.

1.5.3 Projektionen aus anderen Richtungen

Wir gehen weiterhin davon aus, dass wir auf eine Ebene durch den Ursprung projezieren werden (die kann durch Translation erreicht werden), aber diesmal in eine beliebige Richtung schauen.

Wir benötigen hierfür zwei Angaben: Einen Vektor d der die Richtung der Projektion angibt, also nach "hinten" in das Bild zeigt, und einen Vektor vup ("view up"), der die Richtung nach "oben" angibt.

Wir setzen nun $z' = d^0, x' = vup \times z', y' = z' \times x'$ (hierbei bezeichnet d^0 den auf eine Einheit normierten Vektor parallel zu d, also $d^0 = |d|^{-1}d$). Dann bilden x'^0, y'^0, z'^0 ein Orthonormalsystem der Kamerakoordinaten.

Als Matrix für die Koordinatentransformation erhalten wir

$$K = \begin{bmatrix} x'^0 & y'^0 & z'^0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Beachte: x', y' und z' sind hier Spaltenvektoren!

Durch Anwenden von $K^{-1} = K^t$ "drehen" wir die Welt, so dass wir wieder im bereits behandelten Fall der Projektion auf die xy-Achse sind. Bei Projektionsmatrix P für den bekannten Fall ist also die neue Projektionsmatrix

$$P' = P \cdot K^t.$$

Kapitel 2

Light- Shading - Color

2.1 Light

Licht ist es, was eine dreidimensionale Szene realistisch wirken lässt. Moderne Computerspiele, vor allem Ego-Shooter wie aus Half-Life-Serie wären ohne Licht-Effekte nicht annähernd so realistisch und Horror-Spiele wie Doom würden ohne Lichtquellen keine Atmosphäre erzeugen können.
Das Licht in der Realität ist zu komplex, als dass man es bei den meisten dreidimensionalen Anwendungen verwenden könnte, auch wenn man beim Ray-Tracing deutlich näher an reale Lichtquellen herankommt als beispielsweise bei Spielen. In der Realität wird Licht, bzw. die Photonen von der Gravitationskraft beeinflusst, sie werden an den meisten Oberflächen (teilweise) reflektiert, Licht kann gebrochen werden, breitet sich mit sehr hoher Geschwindigkeit aus und erzeugt mit verschiedenen Frequenzen unterschiedliche Farbeindrücke.
Bei der Entwicklung von dreidimensionalen Anwendungen, bei denen Lichteinflüsse berücksichtigt werden sollen, muss man daher die realen Lichtquelle vereinfachen.

2.1.1 Ambient Light

Ambient Light ist das Umgebungslicht, dass alle Objekte im Raum gleichmääig beleuchtet. Es ist in der Regel nicht eindeutig bestimmbar, wo die Quelle für dieses Licht liegt. Es ist einfach da. In den Anfängen der dreidimensionalen Spielen war sie die einzigste Lichtquelle, heute ist das Ambient Light in den meisten Spielen nur sehr schwach, das weitere Licht wird durch seperate Lichtquellen erzeugt.
Die Intensität, mit der alle Objekte beleuchtet werden, setzt sich aus der Intensität des Umgebungslichtes I_{in} und der empirisch bestimmten Materialkonstanten $k_{ambient}$ zusammen:

$$I_{ambient} = I_{in} * k_{ambient}$$

2.1.2 Diffuse Light

Wenn man in einem dunklen Raum eine Kerze anmacht, mit der Taschenlampe durch den Raum leuchtet oder man im Dunklen den Monitor anschaltet: man stellt fest, dass bestimmte Stellen heller sind als der Rest des Raumes, sei es im Kegel des Taschenlampenlichts, um die Kerze oder um den Monitor. Für solche Beleuchtungen verwendet man diffuses Licht, dabei wird das einfallende Licht gleichmäßig in alle Richtungen reflektiert, unabhängig vom Einfallswinkel des Lichtes. Der Einfallwinkel beeinflusst hier lediglich die Intensität der Beleuchtung. Bei dem diffusen Licht hängt die Beleuchtungsstärke von drei Komponenten ab: der Intensität des einfallenden Lichts I_{in}, der Materialkonstanten k_{diffus}, sowie dem Winkel zwischen der Oberflächennormalen N und dem einfallenden Lichtstrahl L mit $\cos\alpha = L * N$. Das heißt, der Term für die diffuse Beleuchtung mit einer Lichtquelle ist:

$$I_{diffuse} = I_{in} * k_{diffuse} * (L * N)$$

2.1.3 Specular Light

Wir können nun unsere Umgebung in ein einheitliches Licht tauchen (ambient light) und einzelne Flächen mit verschiedenen Lichtquelle beleuchten (diffuse light). Doch eines fehlt noch: Glanz. Egal, ob man in die Augen eines Menschen sieht oder eine Stelle eines beleuchtenden Stücks Metall betrachtet: man entdeckt immer wieder so genannte Glanzlichter, d.h. Stellen, die besonders stark beleuchtet sind und daher glänzen. Diesen Effekt können wir bisher noch nicht verarbeiten, aber das wird sich mit Specular Light ändern.
Die Stärke, mit der wir die Glanzstellen sehen, ist hier, anders als bei diffusem Licht, abhängig von der Position des Betrachters. Die Intensität des Glanzlichts setzt sich zum einen wieder aus der Intensität des einfallenden Lichtstrahls I_{in} und der Materialkonstante $k_{specular}$ zusammen. Wie bereits gesagt, ist auch die Position des Betrachters relevant, besser gesagt der Winkel zwischen der Blickrichtung des Betrachters V und der idealen Reflexionsrichtung des ausfallenden Lichtstrahls R: $\cos^n\beta = (R * V)^n$. Da die Stärke des Glanzes auch von der Oberflächenbeschaffenheit abhängt, ist eine zusätzliche Variable n notwendig, die diese beschreibt. Für $n < 32$ ist die Oberfläche rau und für $n > 32$ wäre die Oberfläche glatt. Die vollständige Formel für das Glanzlicht lautet damit:

$$I_{specular} = I_{in} * k_{specular} * (R * V)^n$$

2.1.4 Emissive Light

Ein weiterer bisher nicht behandelter Punkt ist die Eigenhelligkeit. Nehmen wir an, wir befinden uns in einem dunklen Raum und der Computer samt Monitor ist angeschaltet. Wir können die Ausgabe des Monitors deswegen sehen, weil sie eine eigene Lichtquelle darstellt und nicht, wie

zum Beispiel ein Tisch darauf angewiesen ist, dass man sie extern beleuchten muss, damit man es wahrnehmen kann. Doch jedem Monitor, Fernseher, Werbetafel eine eigene Lichtquelle zuzuordnen wäre in vielen Fällen zu rechenaufwendig, weswegen man davon ausgeht, dass solche Objekte, wie der Bildschirm bei Fernsehern und Monitoren eine Eigenhelligkeit besitzen, also ohne Lichtquellen zu sehen sind, selbst aber kein Licht erzeugen. Die Intensitäts für ein Objekt mit Eigenhelligkeit muss dabei entsprechend festgelegt werden.

2.1.5 Phong Illumination Model

Im Juni 1975 stellte der vietnamesische Computergrafiker Bui-Tuong Phong das nach ihm benannte Beleuchtungsmodell vor, das für die Berechnung für die Beleuchtung von Objekten verwendet wird. Sein vorgestelltes Modell ist rein empirisch, das auf keiner physikalischen Grundlage beruht. So wird der Energieerhaltungssatz, der unter anderem besagt, dass ein Objekt nicht mehr Licht reflektieren kann, als von der Lichtquelle ausgesandt wurde. nicht berücksichtigt. Desweiteren ist es recht langsam zu berechnen, aber es findet auch heute noch in vielen dreidimensionalen Darstellungsverfahren Verwendung.
Die Intensität der Beleuchtung setzt sich aus dem Umgebungslicht, dem diffusen Licht sowie dem Glanzlicht zusammen: $I_{out} = I_{ambient} + I_{diffuse} + I_{specular}$

Die einzelnen Komponenten haben wir bereits kennengelernt:
ambient light: $I_{ambient} = I_{in} * k_{ambient}$
diffuse light: $I_{diffuse} = I_{in} * k_{diffuse} * (L * N)$
specular light $I_{specular} = I_{in} * k_{specular} * (R * V)^n$

Zusammengesetzt ergibt sich der Term:
$I_{out} = I_{in} * k_{ambient} + I_{in} * k_{diffuse} * (L * N) + I_{in} * k_{specular} * (R * V)^n$

Was jedoch bei obigem Term noch nicht beachtet wurde, ist, dass es mehrere Lichtquellen für diffuse und specular light geben kann und sich diese Lichtquellen eventuell beeinflussen können, sodass die Lichtintensität steigen kann. Der folgende Term berücksichtigt auch dies und bildet den endgültigen Term für das Phong-Beleuchtungsmodell:

$$I_{out} = I_{in} * k_{ambient} + \sum_{i=0}^{lights}(I_{in} * k_{diffuse} * (L * N) + I_{in} * k_{specular} * (R * V)^n)$$

2.1.6 Light Sources

Bei der Beleuchtung von Objekten oder einer ganzen Szenerie kommen oft eine Vielzahl von Lichtquellen zum Einsatz, im folgenden sollen die gängigsten Lichtquellen vorgestellt werden.

2.1.6.1 Directional Light

Unter direktem Licht versteht man Licht, das gleichmäßig aus einer bestimmten Richtung kommt, jedoch im Grunde keine Lichtquelle hat. Direktes Licht wird beispielweise oft verwendet, um das einfallende Licht der Sonne zu imitieren. Die Lichtintensität ist meist konstant und nimmt auch über große Entfernungen nicht ab.

2.1.6.2 Spot Light

Spot Lights sind Lichtquellen, die, im Gegensatz zu direktem Licht, eine feste Lichtquelle im Raum haben und das Licht kegelförmig in eine besimmte Richtung abgeben. Jedes Spot Light verfügt über zwei Grade an Lichtintensität, einen für das Zentrum, das besonders hell erleuchtet wird und einen für die Beleuchtung des Gebietes um das Zentrum. Spot Lights sind auf Grund der vielen notwendigen Berechnungen die aufwendigsten Lichtquellen, die in 3D-Anwednungen Verwendung finden.

2.1.6.3 Point Light

Punktlichter sind punktförmige Lichtquellen, die das Licht gleichmäßig von ihrer Position aus in alle Richtungen abgeben. In der Regel nimmt die Intensität des Lichtstrahls dabei mit wachsender Entfernung ab. Ein gutes Beispiel hierfür ist sicherlich eine Kerze, die das Licht in alle Richtungen abgibt und zwar gut zum Erhellen eines Tisches, aber schlecht für die Beleuchtung einer ganzen Turnhalle wäre.

2.2 Colour

Wenn wir über Farben, meist im Zusammenhang von Licht, reden, sagen wir im Grunde das Falsche. So ist weißes Licht im Grunde nicht weiß, sondern besteht aus einer Mischung der Spektralfarben, also von Ultraviolet bis Infrarot. Eine Anschauung für diese Zusammensetzung bekommt man beispielsweise, wenn man einen Lichtstrahl in ein Glasprisma fallen lässt. Da die einzelnen Spektralfarben unterschiedlich gebrochen werden, erhält man eine schöne Projektion der Spektralfarben auf ein dahinter stehenden Schirm.
Hermann von Helmholtz entwickelte 1850 die Dreifarbentheorie, nach der man alle Farben, auch Weiß durch das Mischen von drei besimmten Grundfarben erhalten kann. Nach diesem Prinzip arbeiten Monitore und auch Drucker. Wenn man beispielweise mal den Bildschirm oder den Fernseher mit einer Lupe betrachtet, so wird man feststellen, dass an jeder Stelle eines Pixels drei Farbpünktchen sind: je eines rot, grün und blau. Mit diesem RGB-System kann man durch Mischen der drei Farben viele verschiedene Farben erzeugen. Auch heute noch werden meist für einen Farbanteil 8 Bit verwendet, das heißt, man kann mit dem RGB-System 16.777.216 Farben ($= 256^3$) erzeugen kann, was auch als TrueColor bekannt und in den 90ern zum Standard wurde. Heutige Bildbearbeitungssoftware nutzt pro Farbanteil 16Bit,

mit denen man 65.535^3 Farben erzeugen kann. Dies findet beispielsweise für die fehlerfreie übernahme von Kameradaten Verwendung oder minimiert Rundungsfehler bei Filterfunktionen. Ein weiteres, sehr weit verbreitetes Farbsystem ist das CMYK-Farbsystem, bestehent aus den Farben Cyan, Magenta, Gelb und einer Angabe über die Dunkelheit, wobei für letztes meist Schwarz verwendet wird. Die vierte Farbe benötigt man, weil man mit den drei Farben Cyan, Magenta und Gelb wie auch beim RGB-System sehr viele Farbtöne erzeugen kann, aber kein reines Schwarz. Das CMYK-Farbsystem findet vor allem bei Druckern Verwendung und ist dort unerlässlich.

2.3 Shading

2.3.1 Flat Shading

Flat Shading, auch als Constant Shading bekannt, ist ein sehr einfaches und schnell zu berechnendes Schattierungsverfahren. Bei diesem Verfahren wird die Normale eines Polygons ermittelt und an Hand derer werden Lichtwert und Farbe festgesetzt. Dabei erhalten alle Pixel eines Polygons die selbe Farbe und den selben Lichtwert. Verwendet wird dieses Schattierungsverfahren vor allem bei Objekten mit ebenen Flächen wie Quadern, Pyramiden oder Prismen. Für Objekte wie Kegel oder Kugeln ist es hingegeben ungeignet, da es die Oberfläche nicht gleichmääig mit Schatten versehen werden kann und sie daher zu eckig wirkt.
Man könnte Flat Shading jedoch auch für runde Flächen wie bei Kugeln verwenden, allerdings müsste man dafür die Zahl der Polygone stark erhöhen, damit die Oberfläche die eckige Optik verliert, was zu höherem Rechenaufwand führen würde.

2.3.2 Gouraud Shading

Bei dem auch als Farb- oder Intensitätsinterpolations-Shading bekanntem Verfahren werden zu Beginn die Farben an den Eckpunkten des Polygons berechnet und anschlieäend die Vertices als zweidimensionales Bild auf die Bildebene projiziert. Das entstande Abbild des Polygons in der Bildebene wird nun zeilenweise abgearbeitet, wobei die Farben an den Kantenschnittstellen mit der Abtastzeile der Eckpunktfarben interpoliert werden. Aus den Kantenfarben werden dabei wiederum die Farbwerte der Bildpunkte der Abtastzeile interpoliert. Bei der Berechnung der Farbwerte eines Vertex werden dabei die Flächennormalen der angrenzenden Flächen gemittelt. Die letztendlich wiedergegebene Farbe entsteht aus der Lichtquelle, dem diffusen Reflexionskoeffizienten und aus dem Winkel zwischen Flächennormalen und Lichtstrahl zur Lichtquelle.
Gouraud Shading gehört auch mit zu schnellsten Schattierungsverfahren und besitzt im Gegensatz zum Flat Shading keine eckige, abgestumpfte Flächen, sondern der Farb-, bzw. Schattierungsübergang wirkt weich. Allerdings wirkt die Silhouette des Polygons nach wie vor kantig und es kann zu Darstellungsfehlern kommen, beispielsweise Sprünge im Farbverlauf oder ein zu starker Farbkontrast (Machsche Streifen).

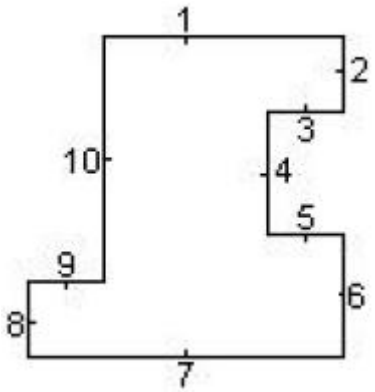
2.3.3 Phong Shading

Wie das Phong Beleuchtungsmodell wurde auch das Phong Shading von Bui-Tuong Phong entwickelt. Beim Phong Shading werden die Normalen an den Ecken des Polygons berechnet und anschließend die Normalen an den Kanten sowie an Kantenschnittstellen interpoliert. Ist dies geschehen, werden anhand der interpolierten Normalen die Farbwerte berechnet, für jedes einzelne Pixel. Das hat zur Folge, dass das Polygon wesentlich ansprechender aussieht, als es bei Gouraud Shading der Fall war, so sind die Kanten wesentlich weicher und die Glanzlichter werden schöner hervorgehoben, aber zu dem hohen Preis, dass die Berechnung wesentlich länger dauert.

2.4 Space Partition

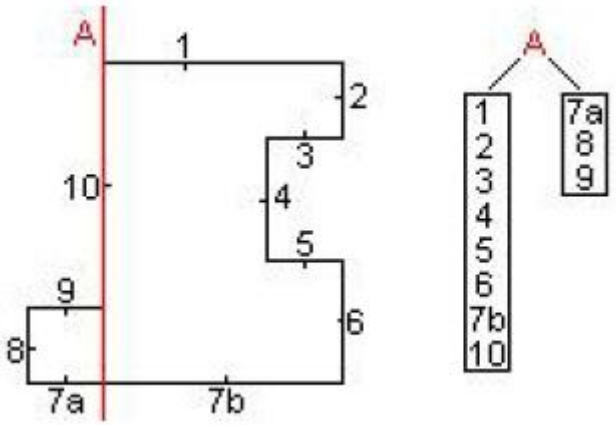
2.4.1 Theorie

Beim Darstellen eines einfachen Objektes, wie zum Beispiel ein Dreieck haben Computer keine großen Schwierigkeiten. Jedoch ist das Rendern von komplexen Objekten wie etwa großen Spielwelten, einem dreidimensionalen Haus oder vergleichbares sehr ressourcenaufwendig und kann bei fehlender Opimierung das reibungslose Laufen einer Anwedung stark beeinträchtigen. Eine mögliche Optimierung besteht im Space Partitioning, zu deutsch: Raumeinteilung. Bei dieser Technik wird überprüft, welche Gebiete des Darzustellenden wirklich berechnet werden und welche nicht. So ist es dem Menschen nicht möglich, ohne sich zu bewegen oder Hilfsmittel zu verwenden, hinter sich zu sehen, weshalb bei einem Ego-Shooter beispielsweise der komplette Levelbereich hinter dem Spiel nicht gerendert werden muss, wenn er nach vorne sieht.
Doch wie funktioniert diese Raumeinteilung genau? Betrachten wir dazu folgende Abbildung:

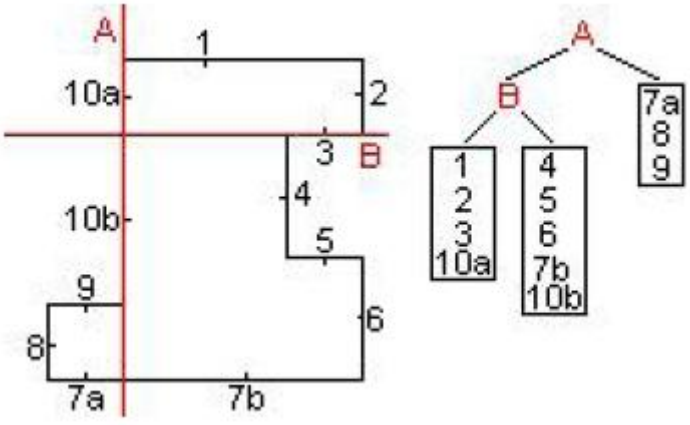


Stellen wir uns vor, die obige Abbildung zeige einen dreidimensionalen Raum aus der Vogelperspektive und wir stehen mit dem Rücken zur Wand 10 und schauen gerade aus auf die Wand 4, die sich genau vor uns befindet. Aus diesem Sichtwinkel ist es nicht möglich die Wände 8, 9 sowie einen Teil von Wand 7 zu sehen. Eine erste sinnvolle Raumunterteilung wäre also den

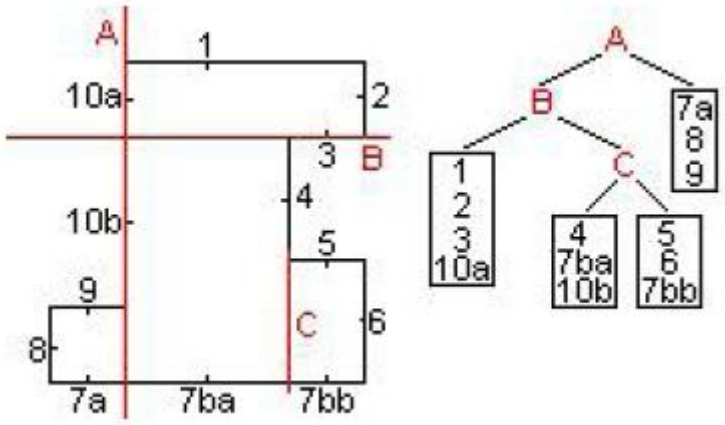
Raum in einen Teil zu gliedern, der vor und ist und dargestellt werden könnte und den nicht sichtbaren Teil hinter uns. Siehe dazu folgende Abbildung:



Wir haben nun eine erste Raumunterteilung, die wir A nennen, sie ist in der obigen Abbildung rot dargestellt. Rechts neben der Abbildung ist eine vorläufie Unterteilung, was vor und was hinter unserer Raumunterteilung liegt. Beachte: jede Wand hat eine Vorderseite, deren Normale durch kurze, schwarze Striche in der Abbildung markiert sind. Fällt eine Trenngerade mit einer Wand zusammen, so übernimmt die Trenngerade die Normale der Wand und die Wand zählt automatisch zur Vorderseite der Trennlinie. Diejenigen Wände die vor der Trenngeraden sind, werden am linken Ast des entstehenden Baumes hinzugefügt und die Wände, die dahinter liegen, an den rechten Ast. Da die Wand 7 nicht eindeutig zu einer der beiden Seite zugeordnet werden kann, muss sie unterteilt werden in 7a, die hinter A liegen und 7b, das vor A liegt. Da der Raum stets in zwei Teile geteilt wird, einem Teil vor und einem nach der Trennlinie, hat jeder Ast maximal zwei äste, es entsteht so ein Binärbaum. Daher nennt man diese Art der Raumunterteilung auch Binary Space Partitioning.
Unser Raum ist aber noch nicht vollständig in konvexe Teile unterteilt, daher ist ein weiterer Schritt notwendig, den man folgender Abbildung entnehmen kann:



Die neue Trennlinie B liegt auf 3 und unterteilt den Teilraum vor A in zwei neue Teilräume, den der vor B und der hinter B liegt. Demenstsprechend formt sich unser Binärbaum auf der rechten Seite der Abbildung weiter. In diesem Schritt muss die Wand 10 gesplittet werden. Es folgt unsere dritte und letzte Unterteilung des gegebenen Raumes:



Die dritte Unterteilung beendet den Unterteilungsprozess nun und wir erhalten 4 konvexe Teilräume. Da die Wand 7 b ein weiteres Mal unterteil werden muss, fügt man dem Namen einen weiteren Buchstaben a,...,z hinzu.
Man könnte auf den ersten Blick meinen, dass man auf Gerade B verzichten könnte, indem man die Teilgerade C einfach nach oben hin verlängert. Auf diese Weise hätte man zwar ebenfalls 4 konvexe Teilräume und einen schöneren Binärbaum, allerdings wäre dann die Darstellung zu wenig optimiert, da die Angabe, dass sich der Spieler hinter C befindet bedeuten würde, dass er sich sowohl im nödlichen wie auch im südlichen Teil des Raumes aufhalten könnte und daher unnötig viel gerendert werden müsste. Noch eine kurze Anmerkung: bei den obigen Abbildungen handelt es sich um zweidimensionale Bilder. Die richtigen Trennungen zwischen den Teilräumen sind im dreidimensionalen Raum Ebenen, die entland der Trenngeraden auf den obigen Abbildungen verlaufen würden.
Binary Space Partitioning, kurz BSP, wird zum Beispiel in der bekannten Ego-Shooter-Serie Quake und vielen weiteren Computerspielen verwendet, da es auf einfache Weise das Rendern Szenerie um einiges beschleunigt. Ein Knackpunkt bei dem BSP-System ist allerdings, dass dynamisch agierende Objekte, wie etwa Monster nicht in das BSP übernommen werden sollte, sondern ausschließlich statische Objekte, wie Häuser oder Städte, da ein Verändern und neuberechnen des BSP-Baumes unter Umständen zu viel Zeit in Anspruch nimmt.

2.4.2 Implementierung

OK, wir wissen also nun was BSP Trees sind und wie man sie erstellt. Doch nun stellt sich die Frage, wie wir sie in unser Programm einbauen können. Da dreidimensionale Grafikprogrammierung in verschiedenen Programmiersprachen betrieben wird, beschränke ich mich hier auf Pseudocode, d.h. Code der keiner Programmiersprache übernommen ist, aber leicht in eine übersetzt werden kann.

2.4.2.1 Baum erzeugen

Nun, wir müssen als erstes ein Polygon als Wurzel wählen. Ob eine Funktion geschrieben wird, die eine ideale Wurzel, wie sie in dem obigen Theorie-Abschnitt vorkam, berechnet oder beliebig gewählt wird, bleibt dem Programmierer überlassen.
Anschließend muss überprüft werden, ob ein Polygon vor oder hinter der Wurzel liegt und je nach dem der entsprechenden Liste zugeordnet werden. Diesen Vorgang muss man hinreichend oft wiederholen, bis der Raum vollständig in genügend kleine konkave Teilräume unterteil wurde. Eine Funktion, die das erledigt könnte wie folgt aussehen:

```
BSPTree *BSPmaketree(Polygon List){
choose a polygon as the tree root
// for all other polygons:
if polygon is in front, add to front list
if polygon is behind, add to behind list
else split polygon and add one part to each list
BSPtree = BSP combinetree(BSPmaketree(front list), root, BSPmaketree(behind list))
}
```

2.4.2.2 Baum durchlaufen

Wir haben nun einen BSP-Baum erzeugt und wollen ihn auch durchlaufen. Dafür werden wir nun eine Funktion DrawTree schreiben, die genau das für uns erledigen wird. Hier muss lediglich überprüft werden, ob der Betrachter vor oder hinter der Wurzel ist und dementsprechend die Komponenten des BSP Baumes berechnen. Zu beachten ist hier allerdings, dass der BSP-Baum immer von hinten nach vorne durchlaufen wird. Das bedeutet, wenn der Betrachter vor der Wurzel ist, erst alles hinter der Trennebene, dann die Wurzel und dann erst die Szene vor dem Betrachter gerendert wird.

```
DrawTree(BSPTree){
if(eye is in front of root){
DrawTree(BSPTree->behind)
DrawPoly(BSPTree->root)
DrawTree(BSPTree->front)
}
else {
DrawTree(BSPTree->front)
DrawPoly(BSPTree->root)
DrawTree(BSPTree->behind)
}}
```

Kapitel 3

Bezierkurven

3.1 Biographie von Pierre Étienne Bézier

Abbildung 3.1: Pierre Étienne Bézier

Pierre Étienne Bézier ist am 1. September 1910 als Sohn und Enkel von Mechanikern in Paris geboren.
Dem folgend beginnt er ein Studium in Maschinenbau, welches er im Jahr 1930 abschließt. 1932 macht er seinen 2. Abschluß in Elektrotechnik.
Mit 23 Jahren kommt er zu Renault, wo er die nächsten 42 Jahre arbeitet. Anfangs nur für einfache Jobs zuständig, arbeitet er sich schnell in die führenden Positionen. 1948 ist er bereits Leiter der Produktionsplanung, 1957 Leiter der Ableitung für Werkzeugmaschinenbau.

Ab 1960 konzentriert er seine Forschungen auf Zeichenmaschinen, interaktive Frei-Hand-Kurven, Oberflächen-Design und computer-gestüztes Design (***CAD***,**C**omputer **A**ided **D**esign).
1968 beginnt Béziers Karriere als Professor an dem „Conservatoire Nationale des Arts et Metiers“, wo er bis 1979 unterrichtet.
Während dieser Zeit schreibt er vier Bücher, diverse Skripte und erhält mehrere Auszeichnungen im Bereich Maschinenbau und CAD.
Am 25 November 1999 stirbt Pierre Étienne Bézier als Ehrenmitglied des **ASME** (**A**merican **S**ociety of **M**echanical **E**ngineers).

3.2 Interpolation und Splines

3.2.1 Freiformkurven

Gegeben seien $(n+1)$ paarweise verschiedene Stützstellen $x_i \in \mathbb{R}$, $0 \leq i \leq n$ und dazugehörige Stützwerte $y_i \in \mathbb{R}$, $0 \leq i \leq n$.

Abbildung 3.2: Punktemenge

In der CAD-Software gibt es zwei Arten der Kurvenformulierung:

- **Interpolation:**
  Gesucht wird eine Funktion f für die gilt:

$$f(x_i) = y_i \text{ für alle } i = 0, 1, \ldots, n$$

- **Approximation:**
  Die Funktion approximiert die Punktemenge, durchläuft sie aber nicht.

Ein linearer Polygonzug wäre der einfachste Fall der Interpolation, jedoch ist der Polygonzug nicht wirklich glatt und er ist nur aus $\mathcal{C}^0$.
Eine weitere Möglichkeit wäre eine Polynominterpolation, die widerum aber bei zu großem n

einen zu hohen Grad hätte und es dadurch zu großen Schwankungen zwischen den Stützstellen kommen kann.

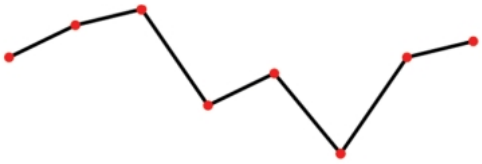


Abbildung 3.3: Linearinterpolation

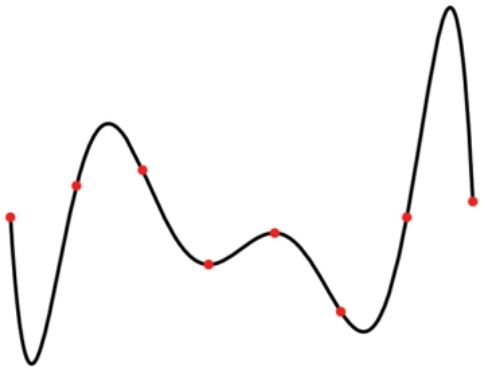


Abbildung 3.4: Polynominterpolation

Eine Verbesserung der Polynominterpolation wäre die Splineinterpolation.

3.2.2 Splines

Definition 1 *Sei* $\Delta := \{a = x_0 < x_1 < \ldots < x_n = b\}$ *eine Unterteilung des Intervalls* $\boldsymbol{I} := [a, b]$. *Dann heißt eine Funktion*

$$S_{3,\Delta} : \boldsymbol{I} \to \mathbb{R} \quad mit \quad S_{3,\Delta} \in \mathcal{C}^2(\boldsymbol{I}) \quad und \quad S_{3,\Delta}|_{[x_i, x_{i+1}]} \in \mathbb{P}_3([x_i, x_{i+1}])$$

ein kubischer Spline. $S_{3,\Delta}(\boldsymbol{I})$ *bezeichnet den Raum aller kubischen Splines auf dem Intervall* $\boldsymbol{I}$. $\mathbb{P}_3([x_i, x_{i+1}])$ *bezeichnet den Raum aller Polynome dritten Grades auf dem Intervall* $[x_i, x_{i+1}]$. *Sind ferner Stützwerte* $y = (y_i)_{i=0,1,\ldots,n}$ *gegeben, so heißt* $S_{3,\Delta,y} \in S_{3,\Delta}(\boldsymbol{I})$ *interpolierender Spline, falls* $S_{3,\Delta,y}(x_i) = y_i$ *für* $i = 0, 1, \ldots, n$ *gilt.*

Durch sie läßt sich eine Funktion finden, die „schöne“ Eigenschaften einer Kurve besitzen und die gegebenen Stützpunkte interpoliert.

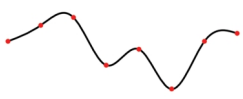


Abbildung 3.5: Splineinterpolation

Jetzt kommen wir zu den Bézierkurven, die zu den approximierenden Kurvenformen gehören.

3.3 Bézierkurven

3.3.1 Motivation

Gegeben seien Punkte P_i mit dazugehörigen Massen $m_i (i = 0, \ldots, 3)$.

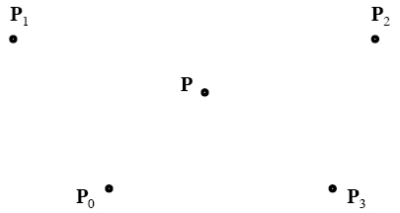


Abbildung 3.6: P_i mit Massen m_i

Dann läßt sich das Massenzentrum P durch folgende Gleichung errechnen:

$$P = \frac{m_0 P_0 + m_1 P_1 + m_2 P_2 + m_3 P_3}{m_0 + m_1 + m_2 + m_3}$$

Statt konstanten Massen m_i betrachtet man sie jetzt als eine Funktion in Abhängigkeit eines Parameters $t \in [0, 1]$. Genauer:

$$m_0 = (1-t)^3 \quad m_1 = 3t(1-t)^2 \quad m_2 = 3t^2(1-t) \quad m_3 = t^3$$

daraus folgt nun für P, da $m_0 + m_1 + m_2 + m_3 = 1$:

$$\begin{aligned} P(t) &= (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3 \\ &= \sum_{i=0}^{3} P_i \binom{3}{i} (1-t)^{3-i} t^i \end{aligned}$$

Die Funktion $P(t)$ beschreibt eine kubische Bézierkurve.

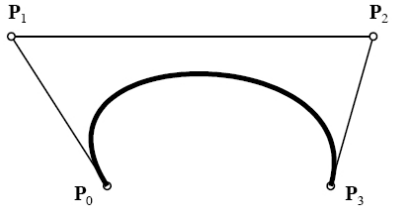


Abbildung 3.7: Kubische Bézierkurve

Die P_i's werden Kontrollpunkte bzw. Bézierpunkte genannt. Diese erzeugen das Kontrollpolygon bzw. Bézierpolgon.
Die m_i's sind Basispolynome, speziell im Fall der Bézierkurven Bernsteinpolynome.

3.3.2 Bernsteinpolynome

Allgemeine Darstellung der Bernsteinpolynome:

$$m_i = B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \qquad (i = 0, \ldots, n)$$

Daraus folgt nun die allgemeine Darstellung einer Bézierkurve vom Grad n mit $n+1$ Kontrollpunkten:

$$P(t) = \sum_{i=0}^{n} P_i \binom{n}{i}(1-t)^{n-i}t^i \qquad t \in [0,1]$$

3.3.2.1 Beispiele

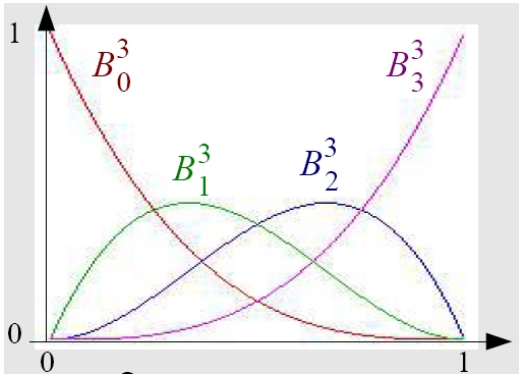


Abbildung 3.8: $n = 3$

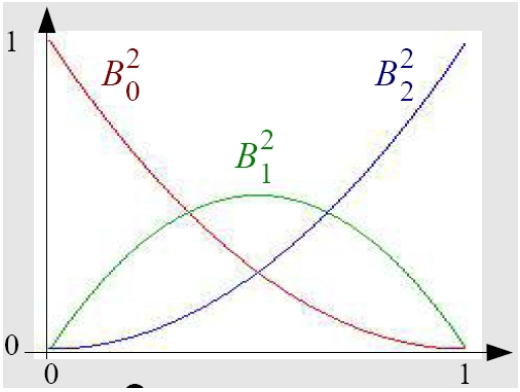


Abbildung 3.9: $n = 2$

n = 3:
$(1-t)^2$
$B_1^3(t) = 3t(1-t)^2$
$B_2^3(t) = 3t^2(1-t)$
$B_3^3(t) = t^3$

n = 2: $B_0^3(t) = (1-t)^3$

$B_1^2(t) = 2t(1-t)$
$B_2^2(t) = t^2$

3.3.3 Eigenschaften der Bézierkurven

- **Zerlegung der Eins:**
  Nach dem Binomialsatz gilt:

$$\sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i = ((1-t)+t)^n = 1$$

- **Symmetrie:**

$$B_i^n(t) = B_{n-i}^n(1-t)$$

- **Positivität:**

$$B_i^n(t) \geq 0$$

- **Globale Abhängigkeit der Bézierkurve:**
  Änderung der Kontrollpunkte verändert auch die Bézierkurve.

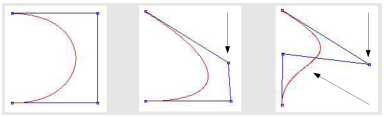


Abbildung 3.10: globale Abhängigkeit

- **Affine Invarianz:**
  Affine Transformationen, wie Verschiebung, Rotation oder Skalierung des Kontrollpolygons wirken gleichermasen auf die Bézierkurve.

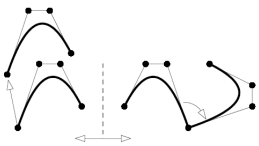


Abbildung 3.11: affine Invarianz

- **Randverhalten:**
  Die Randpunkte der Bézierkurve werden interpoliert und bilden dort jeweils eine Tangente an die Kurve.

- **Konvexe-Hüllen Eigenschaft:**
  Die Bézierkurve liegt immer in ihrer konvexen Hülle, sofern $t \in [0, 1]$.

- **Beschränkte Schwankung:**
  Die Bézierkurve wird höchstens so oft durch eine Gerade geschnitten wie ihr dazugehöriges Kontrollpolygon.

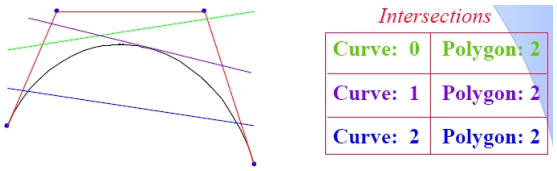


Abbildung 3.12: Beschränkte Schwankung

3.3.4 Anwendung in „font definition“

Um Buchstaben auf dem Bildschirm sowie im Druck ausgeben zu können, behalf man sich früher mit Bitmap-Fonts.
Im Bitmapformat wird jedes Zeichen als Pixelbild gespeichert, was zu Qualitätsverlusten bei höheren Auflösungen geführt hatte.
Durch den Einsatz der Bézierkurven wird es ermöglicht, Schriften auflösungsunabhängig auszugeben.

3.4 Der „de Casteljau Algorithmus“

3.4.1 Idee

Bézier hatte einen mathematischen Zugang zu den Bézierkurven, dagegen bietet der „de Casteljau Algorithmus“ einen geometrischen Zugang.
Er wurde von Paul de Casteljau bei Citroën entwickelt(1910 - 1999).
Der Algorithmus beruht darauf, das eine Bézierkurve geteilt und durch zwei aneinandergesetzte Bézierkurven dargestellt werden kann. Durch rekursive Fortsetzung dieser Teilung nähert sich das Kontrollpolygon der zusammengesetzten Bézierkurven an die Originalkurve an.

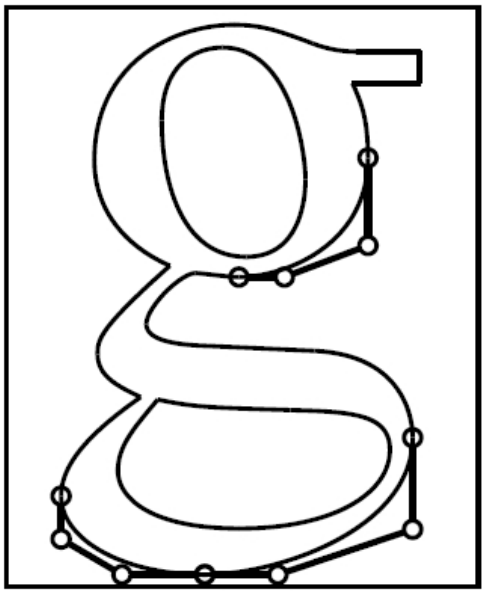


Abbildung 3.13: Font 1

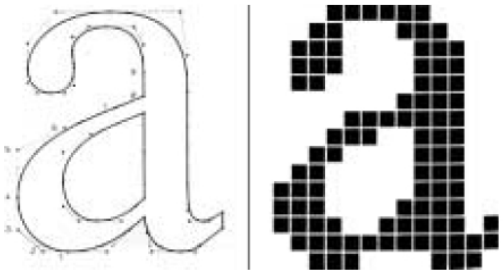


Abbildung 3.14: Font 2

3.4.2 Algorithmus

3.4.2.1 Konstruktion eines Kurvenpunktes

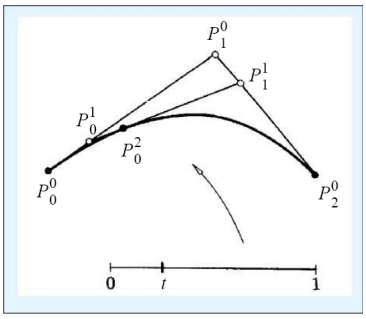


Abbildung 3.15: Quadratische Bézierkurve

Es werden die Strecken $\overrightarrow{P_0^0 P_1^0}$ und $\overrightarrow{P_1^0 P_2^0}$ im Verhältnis $t : (1-t)$ geteilt.
Die aus den durch Teilung entstandenen Punkten P_0^1 und P_1^1 bildende Strecke $\overrightarrow{P_0^1 P_1^1}$ wird wieder im Verhältnis $t : (1-t)$ geteilt. Der nun entstandene Punkt P_0^2 ist genau der Punkt $P(t)$ der Bézierkurve an der Stelle t.
Die allgemeine Rekursionsformel lautet:

$$P_i^{j+1} = (1-t)P_i^j + tP_{i+1}^j$$
$$j = 0, \ldots, n-1 \qquad i = 0, \ldots, n-j-1$$

und es gilt:

$$P(t) = P_0^n$$

Hier ein Beispiel mit $n = 3$ und $t = 0.4$:

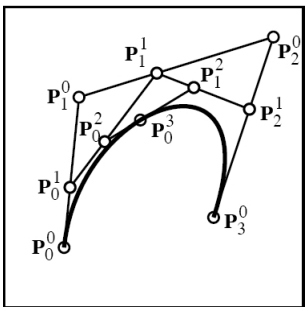


Abbildung 3.16: Kubische Bézierkurve

3.4.2.2 Teilen in zwei Bézierkurven

Anhand der nächsten zwei Abbildungen läßt sich gut die Teilung einer Bézierkurve in zwei aneinandergrenzenden Kurven erkennen.

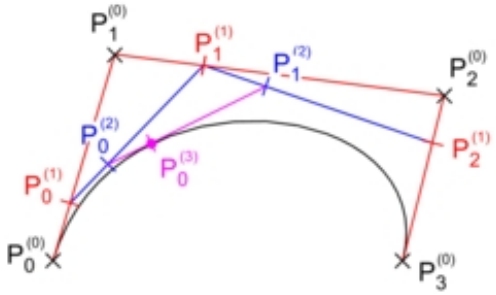


Abbildung 3.17: Teilung

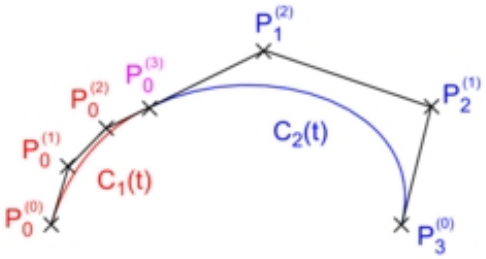


Abbildung 3.18: Zwei Bézierkurven

3.5 Parametrische Ableitung einer Bézierkurve

Die parametrische Ableitung einer Bézierkurve kann geometrisch aus seinen Kontrollpunkten bestimmt werden. Die erste Ableitung einer Bézierkurve vom Grad n mit den Kontrollpunkten P_i kann folgendermaßen als eine Kurve vom Grad $(n-1)$ mit den Kontrollpunkten D_i bestimmt

werden:

$$D_i = n(P_{i+1} - P_i) \qquad i = (0, \ldots, n)$$

Die erste parametrische Ableitung (auch Hodograph genannt) liefert uns den Tangentenvektor. Jedoch ist es wichtig zu betonen, dass diese Gleichung nur für Bézierkurven gilt und nicht beispielsweise für rationale Bézierkurven gilt. Die Abbildung zeigt den Hodograph einer kubischen Bézierkurve an der Stelle $t = 0.3$.

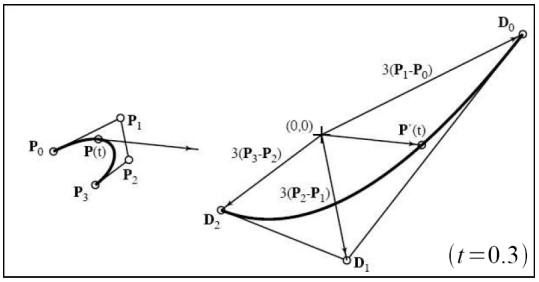


Abbildung 3.19: Parametrische Ableitung

3.6 Kontinuität

Speziell wenn man zwei Bézierkurven aneinanderfügen will, tritt die Frage der Stetigkeit bzw. Kontinuität auf.
Es gibt zwei Typen der Kontinuität, die im Fall der Bézierkurven wichtig sind:

- **Parametrische Kontinuität:**
  Die parametrische Kontinuität ist vergleichbar mit der Stetigkeit aus der Mathematik. Sie wird mit $\mathcal{C}^n$ bezeichnet, wobei n den Grad der Kontinuität angibt:

  $$\mathcal{C}^n : \frac{d^n P_1(u)}{du^n} = \frac{d^n P_2(t)}{dt^n}$$

  $\mathcal{C}^0$ bedeutet also, das die beiden Kurven P_1 und P_2 den selben Endpunkt teilen, $\mathcal{C}^1$ heißt das sie aus $\mathcal{C}^0$ sind und die selben Tangentenvektoren an der Stelle haben, die sowohl in der Länge als auch in der Richtung übereinstimmen.

- **Geometrische Kontinuität:**
  Die geometrische Kontinuität (auch visuelle Kontinuität genannt) ist schwächer als die

parametrische. Sie wird mit $\mathcal{G}^n$ bezeichnet und ist folgendermaßen definiert:

$$\mathcal{G}^n : |\frac{d^n P_1(u)}{du^n}| = \alpha |\frac{d^n P_2(t)}{dt^n}| \qquad \alpha \geq 0$$

$\mathcal{G}^0$ stimmt mit $\mathcal{C}^0$ überein. Auch $\mathcal{G}^1$ stimmt mit $\mathcal{C}^1$ bis auf die Tatsache überein, das der Tangentenvektor beider Kurven an der Stelle dieselbe Richtung hat, aber nicht die gleiche Länge haben muss.

Trivialerweise ist jede Kurve aus $\mathcal{C}^n$ auch aus $\mathcal{G}^n$, aber die Umkehrung gilt nicht.

3.7 „degree elevation“ (Graderhöhung)

Durch die Graderhöhung ist es möglich Bézierkurven vom Grad n durch Bézierkurven vom Grad $> n$ exakt darzustellen.
Die Idee besteht darin, beispielsweise eine kubische Bézierkurve mit $[t + (1 - t)] \equiv 1$ zu multiplizieren.
Diese Multiplikation verändert die Kurve nicht, erhöht jedoch ihren
Grad um 1. Statt den ursprünglichen 4 Kontrollpunkten hat man nun 5.
Die neuen Kontrollpunkte lassen sich wie folgt berechnen:

$$P_i^{\text{new}} = \alpha_i P_{i-1} + (1 - \alpha_i) P_i \qquad \alpha_i = \frac{i}{n+1}$$

Explizit am Beispiel der kubischen Bézierkurve:

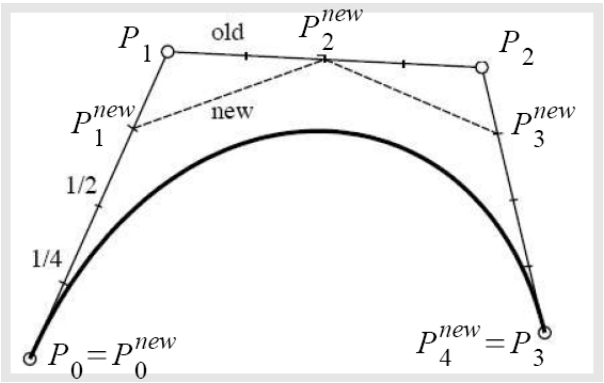


Abbildung 3.20: „degree elevation“

$$P_0^{\text{new}} = P_0$$

$$P_1^{\text{new}} = \frac{1}{4}P_0 + \frac{3}{4}P_1$$
$$P_2^{\text{new}} = \frac{2}{4}P_1 + \frac{2}{4}P_2$$
$$P_3^{\text{new}} = \frac{3}{4}P_2 + \frac{1}{4}P_3$$
$$P_4^{\text{new}} = P_3$$

Auch hier ist erkennbar, das sich das Kontrollpolygon durch rekursive Fortsetzung des Algorithmus der Bézierkurve annähert.

3.8 Rationale Bézierkurven

Rationale Bézierkurven erlauben weitere Darstellungsmöglichkeiten der Kurve. Beispielsweise lassen sich damit Kegelschnitte oder Kreise exakt darstellen. Dies ist dadurch möglich, das eine weitere Variable w_i eingeführt wird, die jetzt die Kontrollpunkte gewichtet.

3.8.1 Gewichte

Die allgemeine Formel einer rationalen Bézierkurve lautet:

$$R(t) = \frac{\sum_{i=0}^{n} P_i w_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)}$$

Aus dieser Gleichung wird klar, woher die rationale Bézierkurve ihren Namen hat. Damit die Basispolynome summiert noch immer die 1 ergeben, muss man sie normalisieren, indem man sie noch durch ihren Gesamtwert teilt.
An der Abbildung läßt sich die Veränderung der Gewichte erkennen:

Eine Gewichtsveränderung ist nicht vergleichbar mit der Veränderung der Kontrollpunkte:

3.8.2 Rationale Bézierkurve als Projektion einer 3D-Kurve

Die Abbildung stellt eine rationale Bézierkurve als Projektion einer 3D-Kurve da. Man stellt sich das folgendermaßen vor:
Es wird eine 2D -und eine 3D-Kurve dargestellt. Die 2D-Kurve liegt auf der Ebene bei $z = 1$, die als Projektion der 3D-Kurve dargestellt wird. Man stellt sich nun einen Kegel vor, der als Spitze den Ursprung hat und als Grundfläche die 3D-Kurve beinhaltet. Jetzt läßt sich die rationale 2D-Kurve als Schnittpunkt zwischen der Ebene bei $z = 1$ und dem Kegel bestimmen.

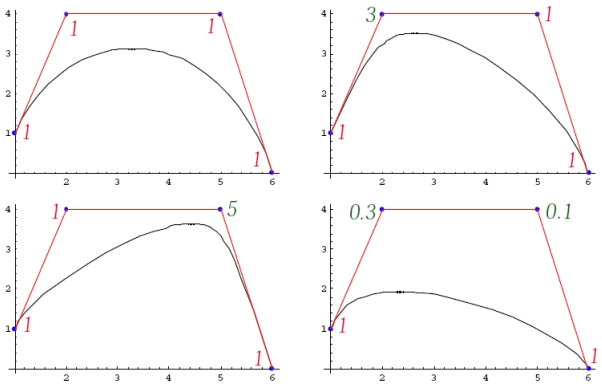


Abbildung 3.21: Gewichtsveränderung

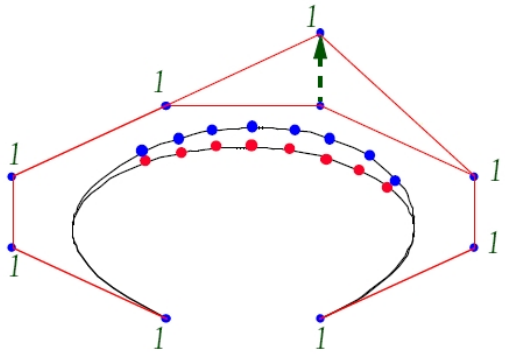


Abbildung 3.22: Veränderung eines Kontrollpunktes

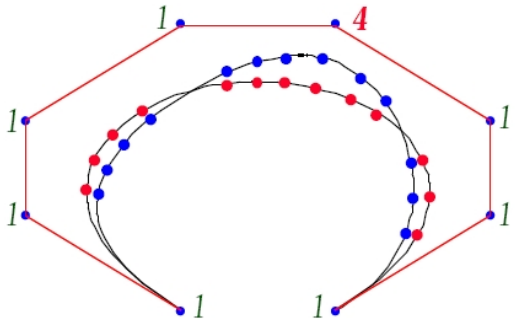


Abbildung 3.23: Veränderung eines Gewichts

3.8.3 Kreisdarstellung

Hier wird anhand einer Abbildung die exakte Darstellung eines Kreises mithilfe der rationalen Bézierkurven demonstriert.

Der komplette Kreis wird durch eine rationale Bézierkurve vom Grad 5 und mit den Gewichten

$$w_0 = w_5 = 1$$

$$w_1 = w_2 = w_3 = w_4 = \frac{1}{5}$$

dargestellt.

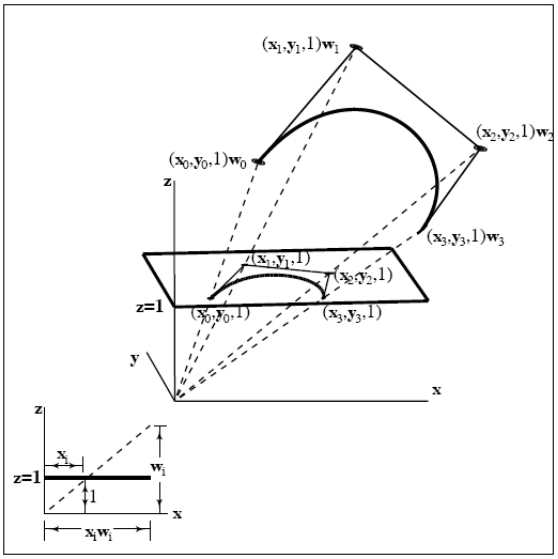


Abbildung 3.24: Projektion einer 3D-Kurve

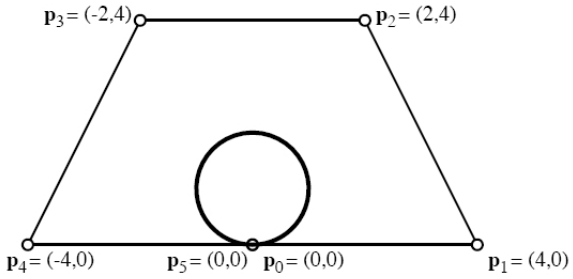


Abbildung 3.25: Darstellung eines Kreises

3.9 Anwendungen

3.9.1 CAD-Bereich

Häufige Anwendung der Bézierkurven finden sich im CAD-Bereich wieder; also speziell in der Autoindustrie dienen die Bézierkurven der Konstruktion diverser Karosserieformen für alle Arten von Automodellen.

3.9.2 Technik

In der Technik dienen die Bézierkurven dazu, diverse Kleinteile, die verschiedene komplexe Konturen aufweisen, digital zu erstellen und letztendlich auch zu konstruieren.

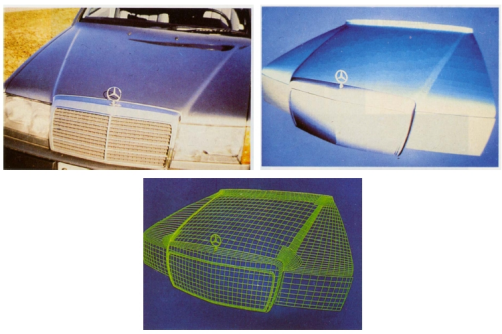


Abbildung 3.26: Anwendung in der Automobilindustrie

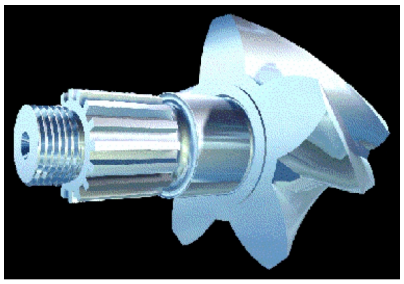


Abbildung 3.27: Anwendung in der Technik

3.9.3 Computer-Graphik

In der Computer-Graphik gibt es eine Vielzahl von Anwendungsmöglichkeiten. Zum einen in der Erstellung von Gesichtkonturen oder in der Landschaftsbildung für die Spieleindustrie. Zum anderen ist es auch möglich mittels den Bézierkurven diverse Bewegungsmodelle zu erstellen, die stetige Verhaltensmerkmale aufweisen.

3.10 B-Splines

Es existieren zwei gravierende Nachteile der Bézierkurven:

- Zum einen die Tatsache, das mehr Kontrollpunkte der Bézierkurve auch eine höheren Grad der Kurve mit sich bringt.

- Und zum anderen, dass die Kurve global abhängig ist, dass heißt eine Änderung eines Kontrolpunktes führt dazu, das sich die ganze Kurve ändert.

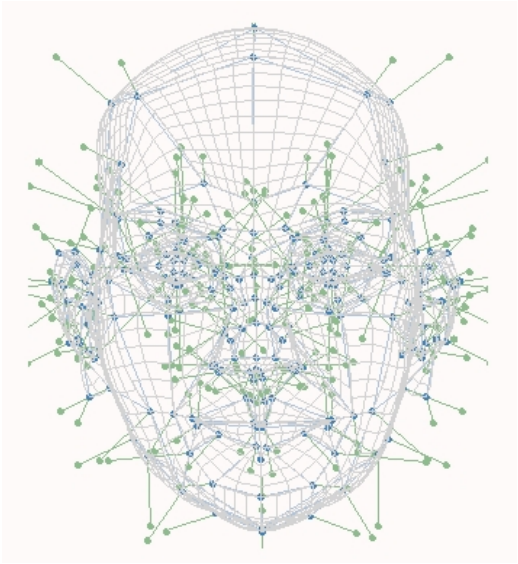


Abbildung 3.28: Gitter-Darstellung (die „Akkupunkturnadeln“ sind die Normalenvektoren)

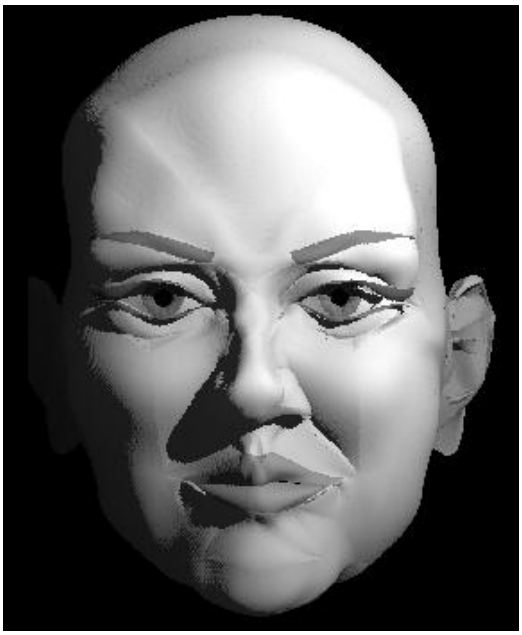


Abbildung 3.29: Endergebnis

Diese Nachteile lassen sich mithilfe der B-Splines beseitigen.

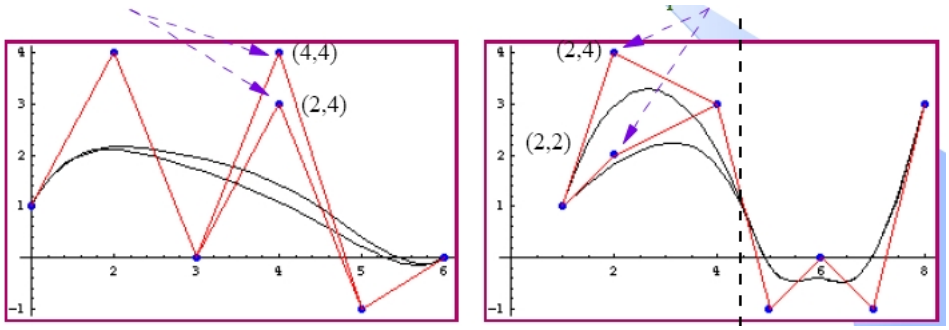


Abbildung 3.30: Unterschied zwischen einer normalen Bézierkurve und einem B-Spline

Ein B-Spline hat die selbe Eigenschaften wie ein kubischer Spline bis auf die Tatsache, das der B-Spline immer nur auf einem beschränkten Intervall definiert ist. Dadurch wird gewährleistet das eine Änderung eines Kontrollpunktes innerhalb dieses Intervalls auch nur die Bézierkurve innerhalb dieses Intervalls verändert. Dies wird durch spezielle Basispolynome ermöglicht, die einen kompakten Träger haben und somit die Kurve nur lokal beeinflussen.

3.11 Bézierflächen

Bisher befanden wir uns nur im $\mathbb{R}^2$. Die Bézierkurven lassen sich auch im $\mathbb{R}^3$ darstellen. Sie werden dort Bézierflächen genannt und haben folgende Gestalt:

$$S(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{ij} B_i^m(u) B_j^n(v) \qquad (u,v) \in [0,1] \times [0,1]$$

$$P_{ij} \in \mathbb{R}^3 \qquad i = 0, \ldots, m \qquad j = 0, \ldots, n$$

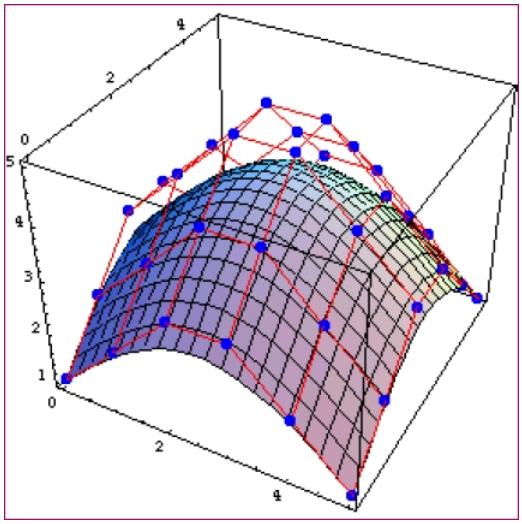


Abbildung 3.31: Bézierfläche

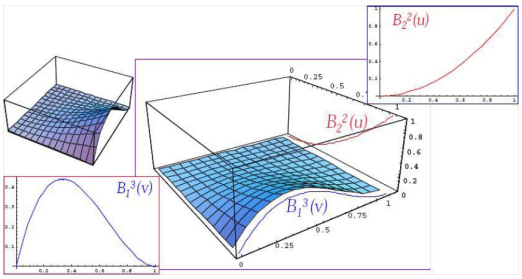


Abbildung 3.32: Bézierfläche im Querschnitt

Kapitel 4

Noise und Turbulence

4.1 Noise

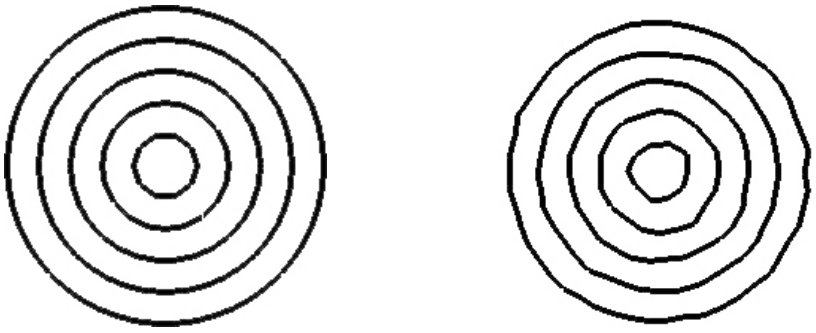
4.1.1 Was ist Noise?

Im Deutschen würde man es als “Rauschen“ bezeichnen. Zu finden ist Rauschen z.B. im Radio, wenn der Sender falsch eingestellt ist, oder auf der Mattscheibe eines älteren Fernsehers, dessen Antennenkabel gezogen wurde (“Ameisenfußball“). Im Alltag begegnet uns “noise/Rauschen“ also als Störung. Allgemeiner: Noise ist eine Zufallsfunktion mit speziellen Eigenschaften.

4.1.2 Wozu kann man Noise verwenden?

Weswegen sollte man nun auf die Idee kommen, eine Störung am Computer künstlich zu simulieren? Nun, es kommt des öfteren vor, dass man auf dem Rechner nicht sterile, exakte, sondern natürlich wirkende Objekte simulieren möchte. Beispiel:

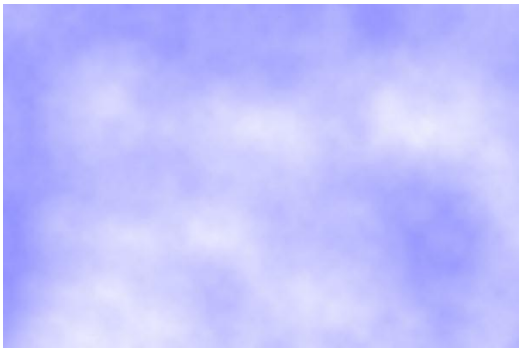
Man möchte eine Zielscheibe (konzentrische Kreise) generieren. Außerdem soll sie aussehen, als sei sie handgemalt.



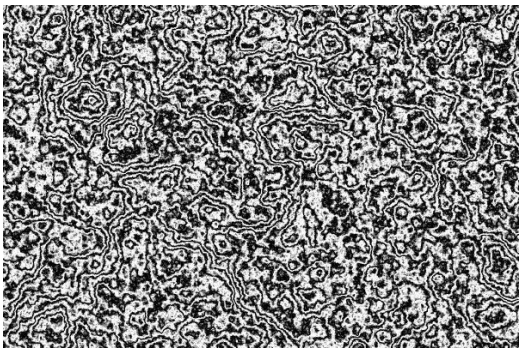
Beide Bilder wurden am Computer erstellt, das erste Bild ohne, das zweite Bild mit “noise“.

Man möchte also Zufälligkeit in seine Programme einbauen, aber nicht vollkommen zufällig, sondern reproduzierbar zufällig. Mit anderen Worten, ich möchte, dass im oben stehenden Beispiel die Linien zwar schief und krumm verlaufen, aber sie sollen beim nächsten Programmstart exakt genauso schief und krumm verlaufen.

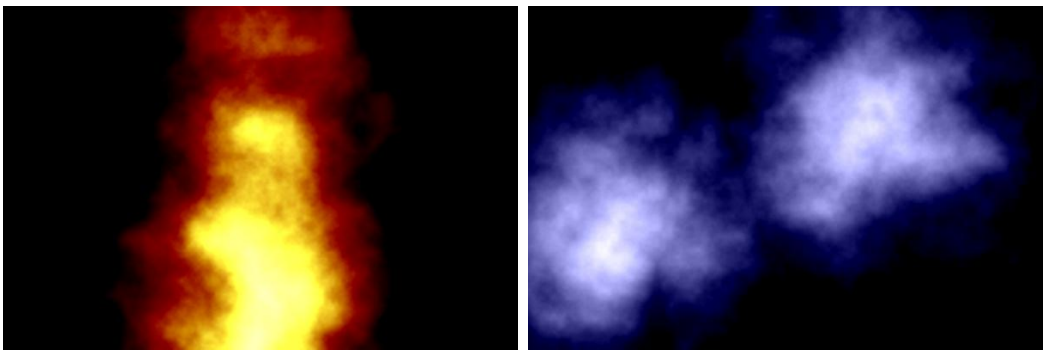
Eine wichtige Verwendungsmöglichkeit ist die einfache Erzeugung von Texturen, beispielsweise Wolken [1]:



Besonders gut darstellen lässt sich eine Marmoroberfläche[2]:



Auch Flammen oder Weltraumnebel lassen sich generieren[3]:



[1] Implementierung siehe „3.1 Wolken“
[2] Implementierung siehe „3.2 Marmor“
[3] Implementierung siehe „3.3 Sterne“ und „3.4 Feuer“

Prinzipiell lässt sich Noise überall dort erzeugen, wo es ein Koordinatensystem gibt. Verwenden kann man Noise für alle Zwecke, in denen man eine Zahl interpretieren kann. (Z.B. als Farbinformation, als Höheninformation, als...)

4.1.3 Welche Eigenschaften muss Noise besitzen?

Zuerst einmal ist Noise eine Abbildung des $\mathbb{R}^n$ auf ein Intervall:

$$noise : \mathbb{R}^n \rightarrow [a, b]$$

- noise sei stetig in allen Komponenten. Andernfalls ergeben sich unnatürlich wirkende Sprünge.
- noise wiederhole sich nicht innerhalb gewisser Grenzen.
- Als Bonus: noise sei differenzierbar (nicht unbedingt notwendig).

4.1.4 Erzeugung von Noise

4.1.4.1 Lattice-Noise

(en. lattice = Gitter) Zu Grunde liegt das Gitter $\mathbb{Z}^n$.$\forall$ z $\in \mathbb{Z}^n$: noise(z) wird zufällig, aber fest gewählt. $\forall$ x $\in \mathbb{R}^n$: Lokalisiere Eckpunkte (z.B. komponentenweise Gaußklammern) und interpoliere mittels dieser Eckpunkte.

Es sind beliebige Interpolationsverfahren möglich. Als Beispiele sei die lineare Interpolation genannt. Sie ist sehr schnell zu berechnen, die Ergebnisse jedoch sehen bescheiden aus. Bessere Interpolationsmöglichkeiten sind die Cosinus- oder die Spline-Interpolation.

4.1.4.2 Gradient-Noise

Auch hier liegt das Gitter $\mathbb{Z}^n$ zugrunde. An jedem Gitterpunkt z $\in \mathbb{Z}^n$ sei noise(z) = 0. Die Gradienten an der Stelle z werden zufällig, jedoch ebenfalls fest, gewählt. Für alle x $\in \mathbb{R}^n$ bestimme man, wie bei Lattice-Noise, die Position im Gitter und ermittle die Eckpunkte der Masche. Für jeden Eckpunkt E_i berechne man das Skalarprodukt $(\nabla E_i, x - E_i)$. Dies gibt den Wert an, den die lineare Funktion mit Gradienten ∇E_i und Stützstelle $(E_i, 0)$ im Punkte x hätte. Hat man für alle Ecken einen Wert errechnet, gewichtet man die Position von x in Bezug auf die Ecken und summiert auf. Ein Beispiel einer Gewichtungsfunktion:

$$\prod_{i=1}^{n} g(x_i) \qquad mit \qquad g(t) = 3t^2 - 2t^3$$

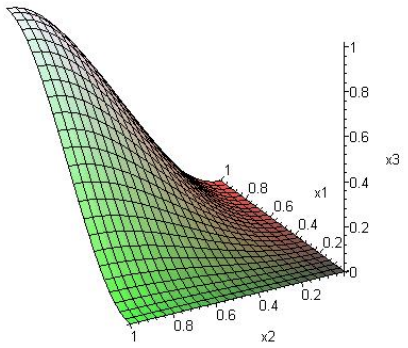


Abbildung 4.1: Gewichtungsfunktion im $\mathbb{R}^2$

4.1.4.3 Beispiel zur Zufallszahlenerzeugung

x = (x¡¡13) ∧ x; return (1.0 - ((x * (x * x * 15731 + 789221) + 1376312589) & 2147483647) / 1073741824.0);

Diese Funktion errechnet aus einer ganzen Zahl mit Hilfe diverser Rechenoperationen (bitwise-AND, bitwise-OR) eine Zahl zwischen -1 und +1. Die auftauchenden Zahlen sind, abgesehen von 1.0, Primzahlen. Durch ihre Verwendung wird die Gefahr einer Wiederholung der Ergebnisse minimiert.

4.1.5 Perlin Noise

Ken Perlin entwickelte Anfang der 80er Jahre des letzten Jahrhunderts das nach ihm benannte Perlin Noise. Der eigentliche Auftrag war es, Computeranimationen für den Film “Tron“ zu erstellen. Als erster und bislang einziger Mathematiker bekam Ken Perlin einen Oskar für einen Algorithmus.

4.1.5.1 Erzeugung der Zufallszahlen

Perlin verwendete keinen Zufallsgenerator im herkömmlichen Sinne, sondern etwas viel Einfacheres:

Man permutiert nach Gutdünken die Zahlen von 0 bis 255 und speichert diese ab. Bei Aufruf des Zufallsgenerators mit Parameter X wird einfach der Wert ausgegeben, der an Stelle X mod

256 gespeichert ist. Der große Vorteil dieses Zufallsgenerators: er ist für die Computergrafik zufällig genug und sehr schnell. Dies wurde auch durch die Verwendung von genau 256 (=2^8) Zahlen erreicht. Da ein Computer Zahlen in binärer Form speichert und verarbeitet, vereinfacht sich die Modulorechnung dahingehend, dass nur die letzten 8 Bits der Zahl betrachtet werden und der Rest ignoriert wird.

4.1.5.2 Noise-Typ

Perlin Noise ist ein Gradient-Noise.

An den Gitterpunkten des $\mathbb{Z}^n$ ist der Wert der Funktion 0.

Es gibt 256 normierte Vektoren, die in einer Liste gespeichert und durch einfaches Nachschlagen erreichbar sind. Die Gewichtungsfunktion ist

$$\prod_{i=1}^{n} g(x_i)$$

$$mit$$

$$g(t) = 3t^2 - 2t^3$$

4.1.5.3 Probleme

1. Die zufällig gewählten normierten Vektoren erfordern die explizite Berechnung des Skalarprodukts. Dies verlangsamt den Algorithmus.

2. Das folgende Bild zeigt deutlich das zweite Problem des Algorithmus:

Abbildung 4.2: Erstellt mit dem Originalalgorithmus

Es bilden sich Klötzchenartefakte. Diese haben ihren Ursprung in der Gewichtungsfunktion:

$$g(t) = 3t^2 - 2t^3$$

$$g'(t) = 6t - 6t^2$$

$$g''(t) = 6 - 12t$$

$$g(0) = 0, g(1) = 0$$

$$g'(0) = 0, g'(1) = 0$$

$$g''(0) = 6, g''(1) = -6$$

Die 2. Ableitung ist an den Maschengrenzen nicht stetig. In einer Masche ist sie -6, in der nächsten +6. Daraus ergeben sich sichtbare "Knicke" im Endergebnis.

4.1.5.4 Lösungen

1. *Verbesserung der Gradientenberechnung* Um die aufwändige Berechnung des Skalarprodukts zu umgehen, verwendet Perlin in seiner aktuellen Version von Perlin Noise nicht mehr 256 verschiedene normierte Vektoren, sondern nur noch zwölf (nicht normierte) Vektoren:

(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),
(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)

Dies sind die Kantenmitten des Einheitswürfels. Da alle denselben Betrag haben ist eine Normierung nicht notwendig. Um die Notwendigkeit einer aufwändigen modulo 12 Berechnung zu umgehen, ergänzt Perlin die zwölf Vektoren um die ersten vier und erreicht so eine Liste von 16 Gradienten, welche wieder bequem durch Betrachtung der letzten 4 Bits einer Zahl implementiert werden kann:

(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),
(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)

Das Skalarprodukt ist nun denkbar einfach. Beispielsweise für den ersten Gradienten, (1,1,0) ergibt sich mit (x_1, x_2, x_3) als Skalarprodukt $1x_1 + 1x_2 + 0x_3$.

Dieses Skalarprodukt lässt sich sogar fest codieren, so dass man keine Gradienten mehr speichern muss, sondern direkte Rechenanweisungen.

2. *Bereinigung der Klötzchenartefakte* Wie oben gezeigt ist die 2. Ableitung der Gewichtungsfunktion nicht durchgehend stetig. Ken Perlin verwendet folgende Gewichtungsfunktion:

$$\prod_{i=1}^{n} g(x_i) \qquad mit \qquad g(t) = 6t^5 - 15t^4 + 10t^3$$

$$g'(t) = 30t^4 - 60t^3 + 30t^2$$

$$g''(t) = 120t^3 - 180t^2 + 60t$$

$$g(0) = 0, g(1) = 0$$

$$g'(0) = 0, g'(1) = 0$$

$$g''(0) = 0, g''(1) = 0$$

Somit ist auch die 2. Ableitung überall stetig. Wie das folgende Bild zeigt, sind keine Klötzchenartefakte zu sehen:

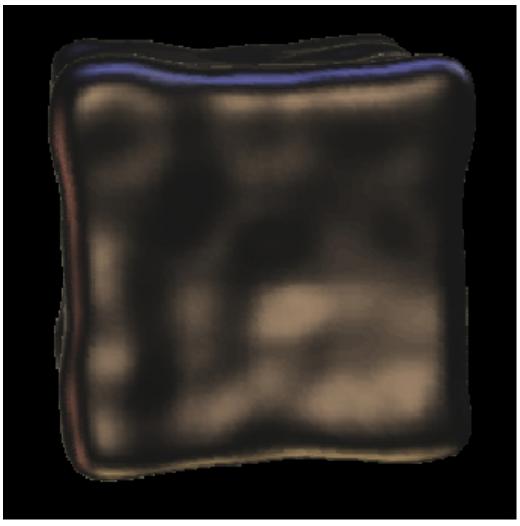


Abbildung 4.3: Erstellt mit dem verbesserten Algorithmus

Zwischenstand

Uns steht eine stetige, sich ausreichend nicht wiederholende Zufallsfunktion zur Verfügung. Dank Ken Perlin ist diese schnell berechenbar. Eine direkte Anwendung dieser Zufallsfunktion, um Natürlichkeit zu simulieren, führt leider nicht immer zum Ziel. Hier der Versuch mit einer Noise-Funktion einen Gebirgszug zu erzeugen:

Das Ergebnis ist mehr als dürftig, man sieht ihm an, dass es von einer Maschine erzeugt wurde. Wenn wir einen genaueren Blick in die Natur werfen, erkennen wir, dass sie äußerst fraktal aufgebaut ist. So wirken die Anden in Südamerika aus dem Flugzeug wie ein Geröllfeld vom Boden aus. Es gibt grobe Unterschiede, wie Gipfel und Täler. Es gibt feinere Unterschiede, wie Felsen und Steine, die wiederum Sprünge und Ritzen haben.

Diese Beobachtung motiviert eine Erweiterung des Algorithmus.

4.2 Turbulence

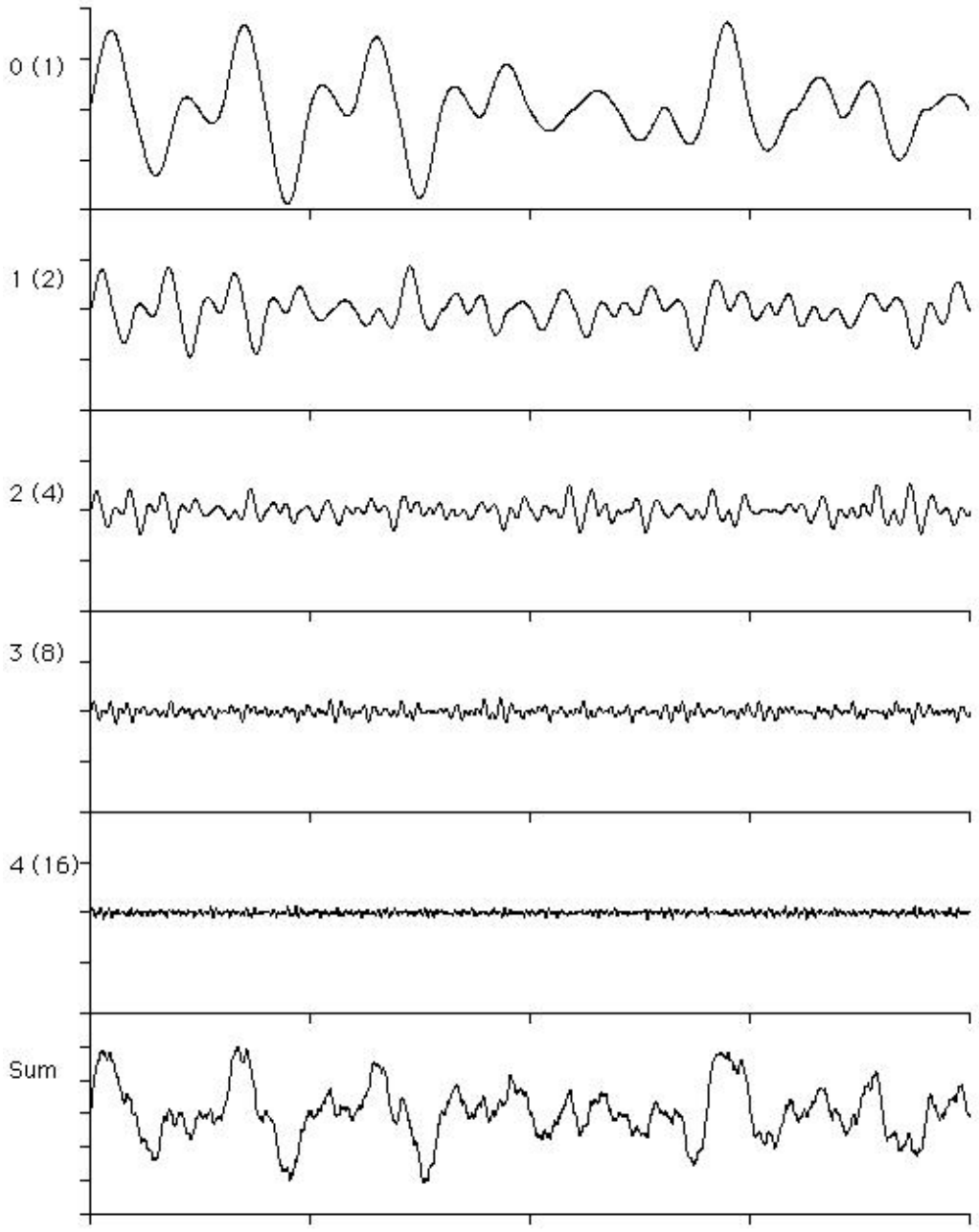
4.2.1 Im $\mathbb{R}^1$

Wir wollen also größere und kleinere Höhenunterschiede erzeugen, die am Ende natürlich aussehen. Dies erreicht man, indem man für jede Unterschiedsstufe eine separate Noise-Funktion verwendet. Diese N verschiedenen Noise-Funktionen werden so gewählt, dass sie in Frequenz (Abstand zwischen zwei Stützstellen) und Amplitude (max(Maximalwert,-Minimalwert)) den zu simulierenden Beschaffenheiten gerecht werden.

In Formeln:

$$Turbulence(x) = \sum_{i=0}^{N-1} \frac{noise(b^i x)}{a^i}$$

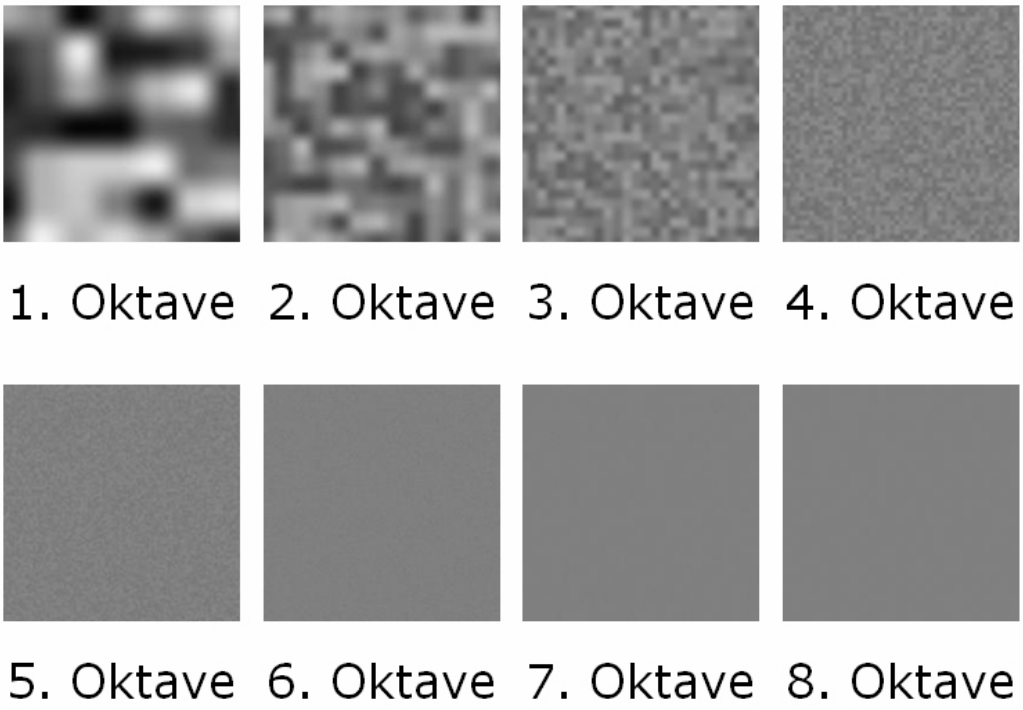
Am häufigsten findet man a=b=2. In Anlehnung an die Musik nennt man die verschiedenen Noise-Funktionen auch “Oktaven“. Im Schaubild:

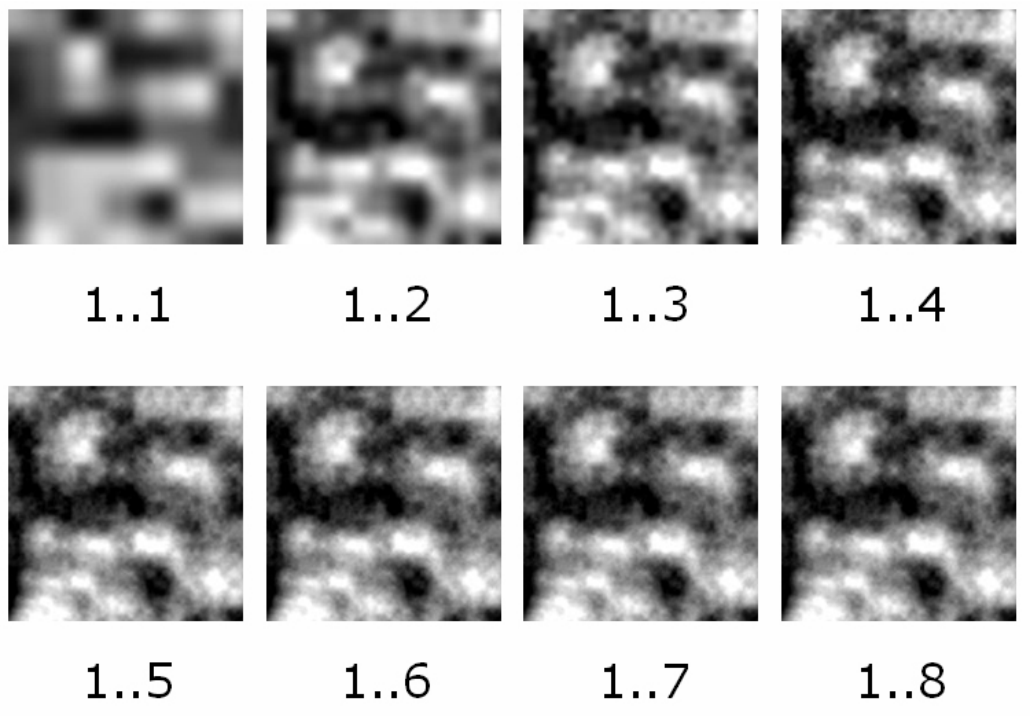


Angewandt auf die Simulation der Silhouette eines Gebirgszugs ergibt sich folgendes Bild:

4.2.2 Im $\mathbb{R}^2$

Zur Verdeutlichung der Wirkung von Turbulence im $\mathbb{R}^2$ hier die einzelnen Oktaven und darunter die Summen:

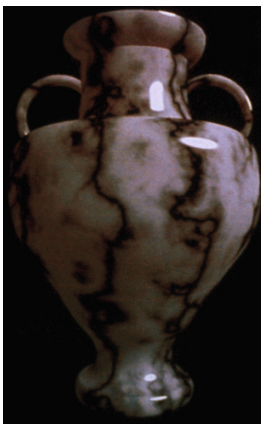




Da die Amplitude in den Bildern 6-8 bereits sehr klein ist wirken die Bilder fast wie eine einheitliche Farbfläche. Dementsprechend gibt es kaum Unterschiede zwischen den Bildern 1..6, 1..7 und 1..8. Praktischerweise bricht man hier die Oktavenberechnung nach Bild 6 ab, da der Unterschied kaum erkennbar ist.

4.2.3 Im $\mathbb{R}^3$

Turbulence im $\mathbb{R}^3$ ist ein mächtiges Werkzeug. Es ermöglicht die Erstellung von dreidimensionalen Texturen, die man wie wirkliches Material bearbeiten kann. Beispielsweise erstellt man einen Block aus „Marmor“ und schneidet dann eine Vase heraus.



Im Gegensatz zum herkömmlichen Texturemapping ist eine derart geschaffene Vase voll interaktiv, d.h. man kann sie zur Laufzeit an jeder beliebigen Stelle zerbrechen und die Bruchstücke wirken, als gehörten sie wirklich zusammen.

Auch zur dreidimensionalen Wolkendarstellung ist Turbulence geeignet. Erweitert man um eine Dimension - die Zeit - so gelingt sogar die realitätsgetreue Darstellung von Wolkenbewegungen. Analog dazu die Darstellung von Staubwolken, Funkenflug, Flammen, Nebel...

4.3 Anwendungen und Implementierung

Die Anwendungsmöglichkeiten sind grenzenlos. überall wo man einen Wert interpretieren kann, ist Noise/Turbulence anwendbar.

Bemerkungen:

Turbulence(**double** X, **double** Y, **int** n)

gibt einen Wert zwischen -1 und -1 zurück. X, Y sind die x_1 bzw. x_2 Werte. n gibt die Anzahl der zu berechnenden Oktaven an.

N wird von Außen eingegeben.

4.3.1 Wolken

Es gibt kaum Objekte in der Natur, die zufälliger anmuten, als Wolken. Durch geschickte Farbwahl und Addieren von Werten zwischen -0.5 und +0.5 erhält man überzeugende Wolkenbilder.

```
void wolken() {
      for (int X = 0; X¡600; X++) {
            for (int Y = 0; Y ¡ 400; Y++) {
            float FARBE = Turbulence(X,Y,N);
            FARBE = (FARBE+1)/4.0;
            punkte[X][Y][0] = .5 + FARBE;
            punkte[X][Y][1] = .5 + FARBE;
            punkte[X][Y][2] = 1;
            }
      }
}
```

4.3.2 Marmor

Marmortexturen weisen eine verzerrte Wellenstruktur auf. Daher liegt es nahe, mit einer Wellenfunktion zu beginnen und Turbulence hinzuzufügen:

```
void marmor() {
      for (int X = 0; X¡600; X++) {
            for (int Y = 0; Y ¡ 400; Y++) {
            double FARBE = (1+sin((X+Turbulence(X,Y,N)*400)*.1))/2.0;
            punkte[X][Y][0] = FARBE; //Die Farbwerte können auch direkt
            punkte[X][Y][1] = FARBE; //verwendet werden
            punkte[X][Y][2] = FARBE;
            }
      }
}
```

4.3.3 Sterne/Nebel

Sterne kann man in der Ebene als Kreis darstellen, dessen Farbe je nach Abstand zum Mittelpunkt errechnet wird. Fügt man dieser etwas Turbulence hinzu, so erhält man Bilder wie unter 1.2 gesehen:

```
void sterne() {
      for (int X = 0; X¡600; X++) {
            for (int Y = 0; Y ¡ 400; Y++) {
            float dist = 1 - sqrt((300-X)*(300-X)+(200-Y)*(200-Y))/300.0*4.0;
            float FARBE = (dist + (1+Turbulence(((300-X*2.0))*2.0,((200-Y*2.0))
            *2.0, N))/2.0)/2.0;
            punkte[X][Y][0] = .8 + FARBE;
            punkte[X][Y][1] = .5 + FARBE;
            punkte[X][Y][2] = FARBE;
            }
      }
}
```

4.3.4 Feuer

Feuer und Flammen sind den Sternen sehr ähnlich, jedoch wird die Flamme indirekt von der Anziehungskraft der Erde verformt. Diesen Effekt erreicht man durch eine Skalierung des x_2-Werts:

```
void flamme() {
	for (int X = 0; X¡600; X++) {
		for (int Y = 0; Y ¡ 400; Y++) {
		float dist = 1 - sqrt((300-X)*(300-X)+.075*(Y)*(Y))/300.0*6.0;
		float FARBE = (dist + (1+Turbulence(((X*2.0)),((Y*2.0)), N))/2.0)/2.0;
		punkte[X][Y][0] = .8 + FARBE;
		punkte[X][Y][1] = .5 + 2*FARBE;
		punkte[X][Y][2] = FARBE;
		}
	}
}
```

Danksagung

Daniel Jungblut
Für die Einführung in C++, OpenGL und LaTeX.

Quellen

http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
http://local.wasp.uwa.edu.au/~pbourke/texture/perlin/
http://student.kuleuven.be/~m0216922/CG/perlinnoise.html
http://mrl.nyu.edu/~perlin/
http://www.noisemachine.com/talk1/

Kapitel 5

Partikelsysteme

5.1 Einleitung

Stößt man beim Modellieren von komplizierten Objekten am Computer an die Grenzen konventioneller Methoden oder will man von Natur aus unscharfe Dinge wie z.B. Feuer darstellen, so greift man zu einem Partikelsystem. Die zugrundeliegende Idee ist, mehrere grafische Primitive (Partikel) so anzuordnen, dass sich ein bestimmtes Bild ergibt. Der Begriff *Partikelsystem* umfasst die Methoden, die notwendig sind, um das gewünschte Ergebnis zu erzielen. Die wichtigsten dabei sind die Generation der Partikel, die Ausstattung derselben mit individuellen Eigenschaften (Farbe, Position, Geschwindigkeit, ...) und die Interaktion der Partikel untereinander und mit ihrer Umgebung. Ein Beispiel: Aus einem Lagerfeuer stoben Funken (Generation), fallen etwas entfernt zu Boden (Schwerkraft) wobei sie sich langsam abkühlen (Farbveränderung) und letztlich nicht mehr sichtbar sind (Elimination). Die Lebenszeit dieser Funken spiegelt den Zyklus wieder, den ein Partikelsystem bei Animationen durchläuft – jeder Partikel wird vom System erfasst, seine Werte modifiziert und die Veränderungen animiert.

5.2 Aufbau und Einsatz

Partikelsysteme haben ein breites Anwendungsgebiet. Sie werden in Computerspielen benutzt um eindrucksvolle Grafikeffekte zu erzielen, in Filmen zur Simulation von Menschenmassen verwendet und sind wichtig zur wissenschaftlichen Visualisierung. Entsprechend schwanken die Anforderungen an ein Partikelsystem je nach Verwendungszweck – jedoch lassen sich einige charakteristische Merkmale heraustellen. Zu einem Partikelsystem gehören mindestens:

Partikelmenge: eine strukturierte Ansammlung von n Partikeln ($n \geq 1$). Die Partikel sind mit Eigenschaften ausgestattet, die sich von Partikel zu Partikel unterscheiden. Durch diese Eigenschaften wird das System im wesentlichen charakterisiert, d.h. welche Art von Eigenschaften benutzt werden, bestimmt den erzielten Effekt.

Systemkern: enthält Methoden, die auf den Partikeln operieren. Hier können Effekte eingebracht werden, die sich auf alle Partikel beziehen sollen, z.B. Schwerkraft. Individualität erhalten die Partikel durch die

Indeterminanz: es gibt einen Zufallsgenerator, der sowohl Vorkommen als auch die Werte der Eigenschaften der Partikel bestimmt. Die stochastische Verteilung sorgt für natürliches Verhalten bzw. Aussehen, innerhalb vorgegebener Grenzen.

Moderne, etwa in Filmen zur realistischen Bilderzeugung verwendete Partikelsysteme sind komplex. Partikel werden nur noch als abstrakte Entität gesehen, welche von ihrer visuellen Erscheinung gelöst ist. Das eröffnet die Möglichkeit, Partikelsysteme auch für Steuerungsaufgaben einzusetzen, wie sie etwa in der Schwarmtheorie nötig sind.

Um größere Vogelschwärme durch eine Schlucht fliegen zu lassen, muss jeder Vogel einen Mindestabstand zum Nachbar einhalten, ohne den Schluchtverlauf zu missachten. Zur Erforschung des Verhaltens von Fischschwärmen sind Partikelsysteme ebenfalls nützlich. Da sich die Fische beim Schwimmen an ihren nächsten Begleitern orientieren, es jedoch keinen herausgestellten Leitfisch gibt, stellt sich die Frage, wie der Schwarm überhaupt eine Richtung findet um etwa in Küstennähe an Plankton zu kommen.

Interaktivität mit und unter Computern ist heutzutage unentbehrlich – und besonders interessant, wenn diese zum großen Teil vom Computer selbst ausgeht. Bei der Steuerung von NPCs (Non Player Characters) in Spielen, vor allem untereinander interagierender Agenten/KIs sind Partikel-, oder in diesem Fall treffender, Schwarmsysteme nützlich.

5.3 Rückblick

William T. Reeves war 1980 bei Lucasfilm Ltd. angestellt und arbeitete in der dortigen Animationssparte, hervorgegangen aus der Arbeit bei Industrial Light and Magic (Star Wars). Heute ist die Firma Pixar mit computergenerierten Filmen erfolgreich. Sie ist 1986 enstanden, nachdem Steve Jobs die Animationssparte von Lucasfilm gekauft hatte. Reeves Aufgabe war es, für den 1982 erschienen Film „Star Trek II: The Wrath of Khan“ eine einen Planeten umlaufende Feuerwand zu animieren. Dafür entwickelte er das erste vollständige Partikelsystem und prägte im Essay über seine Arbeit auch diesen Begriff.

> Particle systems model an object as a cloud of primitive particles that define its volume. Over a period of time, particles are generated into the system, move and change form within the system, and die from the system. The resulting model is able to represent motion, changes of form, and dynamics that are not possible with classical surface-based representations. – Reeves [27].

Die resultierende Visualisierung des Genesis-Effektes, bei dem eine Art Bombe einen Planeten bewohnbar macht, war die erste komplett computergenerierte Szene eines Spielfilms.

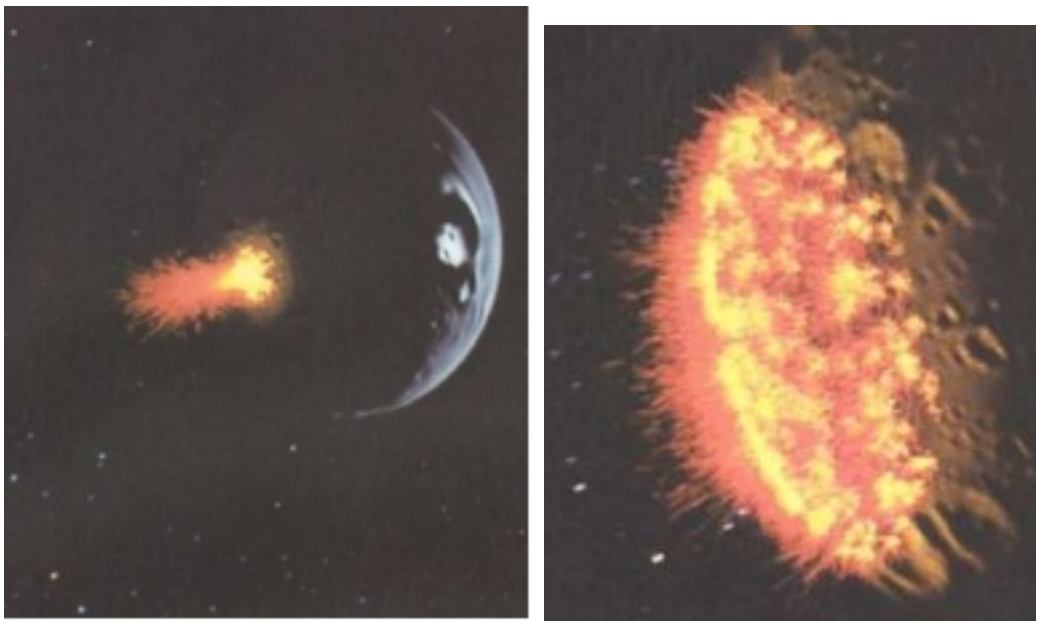


Abbildung 5.1: Der Genesis-Effekt

Vor Reeves war das Konzept bei 2D-Spielen benutzt worden, um die Explosionen kleiner Raumschiffe mit herumfliegenden Pixeln darzustellen. Auch zur Modellierung von einfachem Rauch und großflächigen Galaxie-Ansichten waren Partikel eingesetzt worden, entscheidend war hier jedoch, dass keine Zufallszahlen benutzt wurden. Das System aus Star Trek erhält seine, für damalige Verhältnisse ausreichende Optik, durch die schiere Menge der berechneten Partikel. Es besaß je nach Kameraeinstellung bis zu 750.000 Partikel. Um Aliasing vorzubeugen setzt Reeves „motion-blurring“ ein, was bei so einfachen Objekten wie Raumpunkten einfach anzuwenden war. Beim Motion-blur wird simuliert, wie das Auge eine sehr schnelle Bewegung wahrnimmt, nämlich verwischt.

Durch Zufallszahlen sind Grafikeffekte mit Partikelsytemen auch ohne großen Designaufwand möglich – es muss sich ja niemand um die genauen Ergebnisse kümmern, zum Editieren verändert man nur die Randbedingungen. Das Resultat seiner Animation nennt Reeves „fuzzy object“.

Die Attribute der Partikel die Reeves verwendet hat sind:

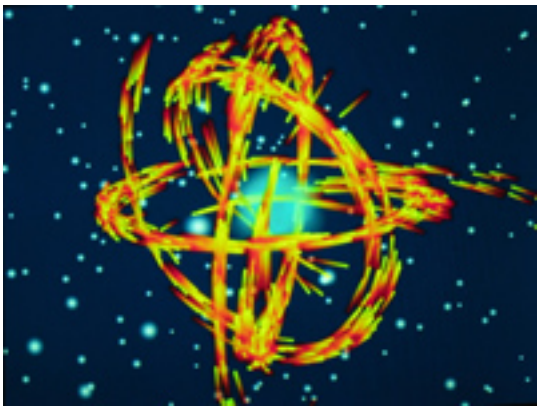
1. Position des Partikels im Raum
2. Beschleunigung
3. Richtungsvektor
4. Farbe
5. Lebensdauer

6. Alter
7. Gestalt
8. Größe
9. Transparenz

Da Position und Geschwindigkeit bzw. Richtung voneinander abhängen, werden sie als Funktionen ihrer ursprünglichen Werte dargestellt. Reeves erwähnt bereits die Möglichkeit, ein System von Differentialgleichungen einzusetzen um das Partikelsystem zu steuern.

Den gesamten Effekt hat Reeves durch mehrere, hierarchisch angeordnete Partikelsysteme erzeugt, indem er die Partikel der unteresten Hierarchie als weiter Partikelsysteme initialisiert hat. Die Feuerwand wirkt so an jeder Stelle des Planeten dicht genug. Zur Sichtbarkeit (Aktivität) eines Partikels zählt neben den Werten „Alter“ und „Lebensdauer“ noch die Position, denn mit dem Planeten kollidierende Teilchen wären nicht mehr sichtbar.

Das Rendern geschah unter jedoch einigen vereinfachenden Annahmen. Es gab keine Kollisionserkennung, der Planet wurde in die Szene geschnitten. Die Belichtung wurde vereinfacht, indem die Feuerpartikel selbstleuchtend gerendert wurden. Das führte zu dem hellen (heißen) Kern der Explosion, da sich hier die Farben vieler Partikel überlagern.



Szene aus „Particle Dreams“

Ein weiterer Meilenstein auf dem Weg zu heutigen, war das von Karl Sims 1988 verwendete Partikelsystem. Seine Video-Demonstration „Particle Dreams“ wurde auf einem CM-2 Supercomputer berechnet. Mit ca. 65.000 1-Bit CPUs konnte sich jeweils eine CPU um die Bahnen eines Partikels kümmern. Das war auch sinnvoll, denn es wurden alle physikalischen Einwirkungen parallel simuliert – auch Wechselwirkungen der Partikel untereinander. Das Video sieht, die „große“ Rechenleistung bedenkend, noch sehr nach 1980 aus – für heutige Verhältnisse (im Vergleich zu heutigen Supercomputern) war der CM-2 ein einfacher Abakus. Anschauen lohnt sich trotzdem, denn man erkennt sehr gut die Vielfältigkeit, mit der ein Partikelsystem einsetzbar ist. Man kann das Video im Internet finden.

(Spiel-)Filme sind heute oft mit Computergrafik aufgebessert. Ein bekanntes Beispiel aus der jüngeren Vergangenheit ist die Verfilmung der „Heer der Ringe“ Trilogie. Es wurde eigens ein System programmiert um die epischen Schlachtszenen lebensecht zu gestalten. Jeder der tausenden Soldaten ist dabei autonom in seinen Handlungen (solange er nicht versucht zu fliehen). Bei Kamerafahrten über das Schlachtgeschehen ensteht so keine Handlungsuniformität.

5.4 Struktur

In diesem Abschnitt wird der Aufbau eines Partikelsystems näher beleuchtet. Die Struktur der einzelnen Komponenten bezieht sich auf Witkin [28].

Partikelsysteme sind dynamische Systeme, deren Ergebnis ein Phasenraum ist. Ein Phasenraum wird durch Variable und ihren zeitlichen Ableitungen aufgespannt, welche in unserem Fall die Position und Beschleunigung der Partikel betreffen. Um natürliche Simulationen zu erreichen, wird die Bewegung der Partikel nach dem physikalischen Grundsatz

$$\vec{F} = m * \vec{a}$$

berechnet. Die zweite Ableitung der Ortsvariablen x entpricht der Beschleunigung, so erhält man die Differentialgleichung zweiter Klasse

$$\ddot{x} = \vec{F}/m$$

welche durch die Einführung von v als Maß der Geschwindigkeit in das Gleichungspaar

$$\dot{v} = \vec{F}/m$$

$$\dot{x} = v$$

umgewandelt wird (Die erste zeitliche Ableitung des Orts ergibt die Geschwindigkeit). Im dreidimensionalen Raum fasst man nun die Komponenten des Orts- und Geschwindigkeitsvektors zu einem sechs-Komponenten Vektor zusammen und erhält die Differentialgleichung

$$[\dot{x_1}, \dot{x_2}, \dot{x_3}, \dot{v_1}, \dot{v_2}, \dot{v_3}] =$$

$$[v_1, v_2, v_3, f_1/m, f_2/m, f_3/m]$$

Das Partikelsystem kann also als ein $6n$ dimensionaler Phasenraum betrachtet werden, mit n als Anzahl der Partikel. Jede Punktkombination gibt einen Zustand des Partikelsystems an, es wird also durch den Phasenraum vollständig charakterisiert.

Etwa ab 25 FPS (Frames Per Second/Bildern Pro Sekunde) wirken bewegte Bilder wie eine flüssige Animation. Ein Partikelsytem durchläuft im Normalfall also mindesten 25 Zyklen pro Sekunde. Am Ende jedes Zyklus wird aus den verändertern Partikelvektoren und Attributen ein neues Bild erstellt. Der Zyklus besteht aus

Emitter: Auf Englisch auch oft als Canon bezeichnet. Jeder Zyklus startet mit dem hinzufügen neuer Partikel, sofern dies notwendig ist um den Verlust der Partikel auszugleichen, deren Lebendauer abgelaufen ist. Natürlich bieten sich hier alle Möglichkeiten die Partikelmenge zu kontrollieren. Um beispielsweise einen Geysir, der nur alle paar Sekunden/Minuten/Stunden, aktiv ist darzustellen wir die Generation phasenweise ausgesetzt. Vom Emitter werden auch die Anfangswerte gesetzt, er enthält also schon einen Zufallsgenerator.

Animator: In der Animator-Phase werden die Partikel-Eigenschaften modifiziert. In dieser essentiellen Phase wird die neue Position der Partikel bestimmt, zu alte Partikel entfernt, neue Farbwerte vergeben oder Texturänderungen durchgeführt. Wieviel Kontrolle hier implementiert wird liegt im Ermessen der Entwickler. Kraftfunktionen wie Schwerkraft, Querruck oder Interaktion der Partikel bspw. mit dem Mauszeiger des Benutzers müssen hier berücksichtigt werden. Witkin sieht eine Liste von Kraftfunktionen vor, die nacheinander auf ein „*force*“ Attribut angewendet werden.

Collider: In diesem Schritt, der wohl der rechenaufwändigste ist, wird durch einen Algorithmus überprüft, welche Partikel mit welchen Elementen in Wechselwirkung treten. Werden nur Kollisionen mit (planen) Oberflächen berücksichtigt, reicht es einen Normalenvektor $\vec{N}$ und einen Punkt P auf der Fläche zu kennen. Das Ergebnis von $(X-P)*N$ (X ist Position des Partikels) gibt dann an, ob sich der Partikel vor der Ebene (> 0), auf der Ebene (0) oder schon hinter der Ebene (< 0) befindet. Bei Partikel-Partikel Kollisionen wächst die Zahl der nötigen Vergleiche drastisch an. Noch aufwändiger wird es bei Partikeln mit unsymmetrischer Form.

Renderer: Die Partikel werden an den von ihnen eingenommen Koodinaten in einem spezifizierten Framebuffer gerendert. In Szenen in denen das Partikelsystem nur ein Element unter vielen ist, kann ein Sprite als Render-Ebene dienen. Raytracing ist bei Partikelsystemen sehr aufwändig, erhöht aber die Qualität enorm. Bei Echzeitanwendungen wird man sich Mühe geben, den Renderschritt so effektiv wie möglich zu Implementieren, es werden deshalb Annahmen gemacht, die den Algorithmus vereinfachen. Bereits angesprochen waren die Möglichkeit, Kollisionen zu ignorieren, so muss der Algorithmus nur die Partikelmenge durchlaufen, und das Eigenlicht der Partikel um die Beleuchtung von aussen zu sparen. Weitere Optimierungsideen sind Datenstrukturen die effizient durchlaufen werden können (Bäume), Wiederverwendung von Partikeln oder die bewusste Beschränkung der absoluten Anzahl an Partikeln.

Ein hoher Detailgrad wird immer auf Kosten von Rechenzeit erkauft. Bei dynamischen Partikelsystemen in Echtzeitanwendungen, die Feuer, Wind o.ä. simulieren (was oft in Spielen der Fall ist), reicht die heute verfügbare Rechenzeit nicht aus um realitätsechte Bilder zu produzieren. Aus diesem Grund gibt es die Tendenz Computer mit Zusatzchips auszurüsten, deren spezielle Aufgabe die Lösung großer Gleichungssysteme ist. So wird die CPU entlastet.

Mit Partikelsystem generierte Pflanze

5.5 Ein einfaches Partikelsystem in C

Wie die Implementation in C ungefähr ablaufen kann wird dieser Abschnitt beschreiben. Ich habe, um die angesprochene Struktur darzulegen, ein möglichst einfaches Partikelsystem in C geschrieben. Ich gehe hier kurz auf die Datenstrukturen und Programmaufbau ein.

Die Grafikkarte wird über die verbreitete SDL (Simple Direct Media Layer) Bibliothek angesprochen. Diese bietet Routinen um direkt in den Grafikspeicher zu schreiben. Das Programm besteht aus einer einzigen Quelldatei und kompiliert (sollte kompilieren) auf den von SDL unterstützten Betriebsystemen. Der Quelltext enthält Informationen zur Installation und ist gut dokumentiert. Der Quelltext ist auf der oben angegebenen Website bereitgestellt.

Ein Partikel wird durch

```
typedef struct {
  unsigned int pos[2];
  int vel[2];
  Uint32 color;
  float lifetime;
  Particle *next;
} *Particle
```

definiert. Da das System 2D arbeitet und ein Partikel als ein Pixel gerendert wird, ist seine Position als Koordinate angegeben. Bei SDL wird der Buffer von links oben aus angesprochen, dort liegt der Nullpunkt. Das Fenster (der Buffer) wird nach rechts und nach unten aufgespannt. Im zweiten Array ist ein Punkt angegeben der relativ zur Position den Richtunsvektor aufspannt. Je länger der Vektor (d.h. je größer der euklidische Abstand), desto schneller fliegend wird der Pixel animiert. Farbe und Lebensdauer sind selbsterklärend. Der Zeiger auf den nächsten Partikel dient dem Aufbau einer verlinkten Liste, die alle Partikel enthält.

Das Partikelsystem (Der Kern) sieht wie folgt aus:

```
typedef struct {
```

```
  Particle *list;
  unsigned int n;
  float t;
} *ParticleSystem
```

Der Zeiger wird zum Zugriff auf das erste Element der Partikelliste benötigt. Der Integerwert n ist die aktulle Anzahl der Partikel im System. Die Fließkommazahl t zählt die Ticks (Zeiteinheiten) die das System aktiv war.

5.6 Fazit

Partikelsysteme sind zu einem Universell einsetzbaren Konzept geworden. Das seit den achtziger Jahren drastisch gewachsene Leistungsvermögen von Computern macht Programme mit mehreren 10.000 Partikeln auch auf Heimcomputern möglich. Umgekehrt üben Entwicklungen gerade im Bereich Computergrafik weiter großen Druck auf die Hardware aus, den steigenden Anforderungen gerecht zu werden. Partikelsysteme stehen hier in vorderster Reihe. Es ist noch Gegenstand aktueller Entwicklungen effiziente und hoch realistische Simulationen mit Partikelsystemen durchzuführen.

Kapitel 6

Autonome Agentensysteme

6.1 Einleitung

Sehr geehrte/r Leser/in, mit dieser Arbeit möchte ich versuchen, Ihnen das Steuerverhalten von so genannten autonomen Charakteren (oder Agenten) zu erklären. Diese Charaktere kann man sich am besten als so genannte NSC (Nicht-Spieler Charakter) vorstellen, bekannt geworden aus Computer-/Consolenspielen. Im Hauptsächlichen werde ich versuchen, Ihnen Ansätze zu geben, um in einem Programm Objekte selbstständig zu bewegen. Dieses wird kein Programmierkurs, deswegen sollten Programmierkenntnisse vorhanden sein, sofern Sie die Absicht haben, mit dieser Arbeit einen Agenten programmieren zu wollen.

6.2 Was sind Agenten überhaupt?

Wir beginnen mit einem kleinen Rückblick, damit man einen Einblick in einen autonomen Charakter oder kurz *Agent* bekommt. Wörtlich übersetzt bedeutet Agent lediglich, dass es eine Person ist, die für eine andere Person agiert. Diese Bedeutung trifft es schon recht genau, jedoch ist dieses sehr weit gegriffen und wir wollen uns auf die Agenten der Computersprache konzentrieren: Grundsätzlich unterteilt man solch einen Agenten in drei Kategorien, welche ich an einem Beispiel versuche zu erklären: Die Situation: Eine Straße führt von A nach B und zwischen A und B steht ein Hindernis. Unser Agent startet von A und soll nach B fahren.

6.2.1 Reflex-Agent:

Dieser Agent braucht eine Aktion, worauf eine Reaktion folgt (so genannte Wenn-Dann Abfragen). In unserem Beispiel würde der Agent also die Straße entlangfahren und falls wir eine Wenn-Dann Abfrage eingebaut haben (z.B. Wenn Hindernis weniger als 30 Meter entfernt, dann weiche aus), würde unser Agent ausweichen und vom Kurs abgekommen sein.

6.2.2 Ziel orientierter Agent

Bei dieser Art des Agenten ist eine Rückkopplung erlaubt. Falls man also das Hindernis umfahren hat, würde der Agent von der neuen Position aus versuchen, das Ziel(Punkt B) zu erreichen.

6.2.3 Nutzen maximierender Agent

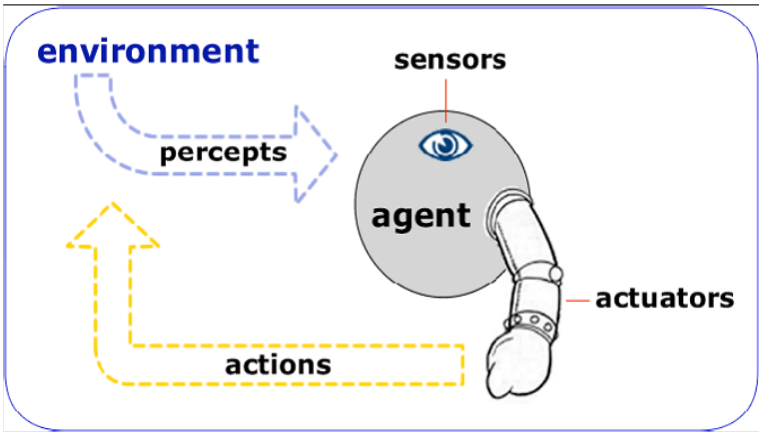
Im Gegensatz zum Ziel orientierten Agenten würde dieser sich auch mit Teil-erfolgen zufrieden geben. Falls Punkt B nach dem Ausweichen nicht mehr erreichbar wäre, würde dieser Agent versuchen, zu einem neuen Punkt C zu gelangen, welcher nächstmöglich an C liegt. Da der Ziel orientierte Agent nur Erfolg oder Misserfolg kennt, würde dieser nichts mehr tun, da das Ziel offensichtlich verfehlt wurde.

Hiermit sollten die drei grundlegendsten Agenten-Typen erklärt sein.

6.3 Der Aufbau

6.3.1 das Äußere

Wir stellen uns den Agenten am Anfang als beliebiges Objekt vor, welches wir vom Programm folgendermaßen aufteilen:



Hier erkennen wir noch einmal genauer, wie ein Agent funktionieren soll. Es gibt äußere Einflüsse, also eine Entwicklung der Dinge (Environment), welche unser Objekt mit bestimmten Abfragen (Sensors) wahrnehmen kann. Je nach Programm handelt der Agent (actuators), und etwas Neues geschieht und nimmt Einfluss auf die Umwelt (actions).

6.3.2 das Innere

Gehen wir in den Agenten hinein und betrachten den inneren Aufbau. Dieser besteht im allgemeinen wieder aus drei hintereinandergeschalteten Funktionen.

1. *Action Selection:* Nachdem die Situation erkannt wurde, beginnt die Auswahl der möglichen Handlungsweisen. Es wird eine mögliche Handlung ausgewählt und weitergegeben. Im allgemeinen wird hier die bestmöglichste Strategie entwickelt. Im Programm wäre es also die Auswahl der Funktion, die aufgerufen werden soll.

2. *Steering:* Das wichtigste Element, worauf sich der weitere Vortrag auch beziehen wird. Hier wird der bestmöglichste Pfad gefunden und weitergegeben.

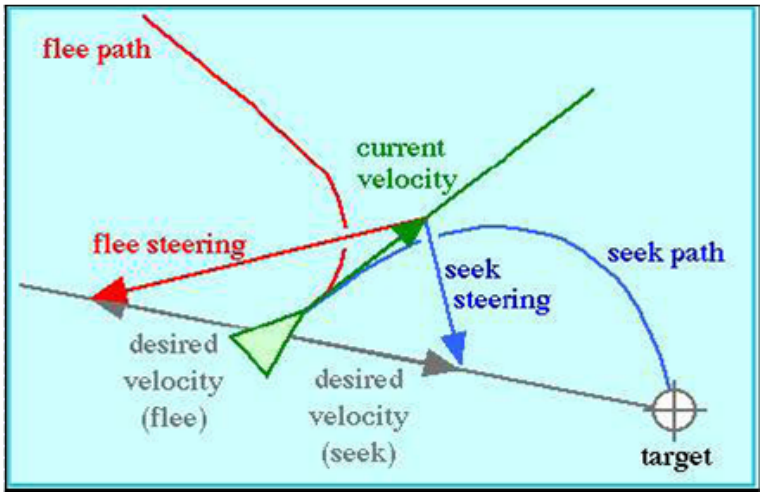
3. *Locomotion:* Dieses beschreibt letztendlich die Animation, die der Betrachter auf dem Bildschirm sieht. Sei es einfach nur ein sich bewegendes Objekt oder eine komplexere handelnde Grafik. Es bleibt dem Programmierer überlassen, was er hier einbinden möchte.

6.4 *Steering Behaviors*

Wenden wir uns nun dem *Steering(3.2.2)* etwas mehr zu. Wir fragen uns natürlich, was genau können wir tun, um den bestmöglichsten Pfad zu finden. Dieses soll nun anhand verschiedener Situationen erklärt werden.

6.4.1 Suchen, Fliehen und Ankommen(„Seek, Flee and Arrival")

Hier handelt es sich um das so genannte Suchen eines Objektes oder Fliehen vor einem Objekt. Allein mit dieser Funktion lässt sich viel anfangen.



Am Anfang scheint diese Grafik recht unübersichtlich zu sein, aber betrachten wir vorerst nur unser Objekt und den davon ausgehenden grünen Pfeil. Dieses stellt den momentanen Bewegungsvektor dar. Würde unser Objekt stillstehen, könnte er ja auch nichts verfolgen. Also geben wir ihm eine bestimmte Bewegungsgeschwindigkeit, die wir in diesem Vektor festhalten.

Betrachten wir nun auch den grauen Pfeil, der in Richtung des Objektes zeigt, den Seek-Vektor. Anhand der Größe dieses Vektors können wir dann festlegen, wie schnell unser Agent auf das Ziel zusteuern soll.

Anhand des Bewegungsvektors und des Seek-Vektors können wir nun unseren blau eingezeichneten Steering-Vektor berechnen, damit der Agent dem blauen Pfad folgen kann.

Im Programm würde dieses etwa so aussehen:

$$gew\_geschw = normalize(aktuelle\_richtung - seek) \cdot max\_speed$$

$$steering = gew\_geschw - aktuelle\_geschw$$

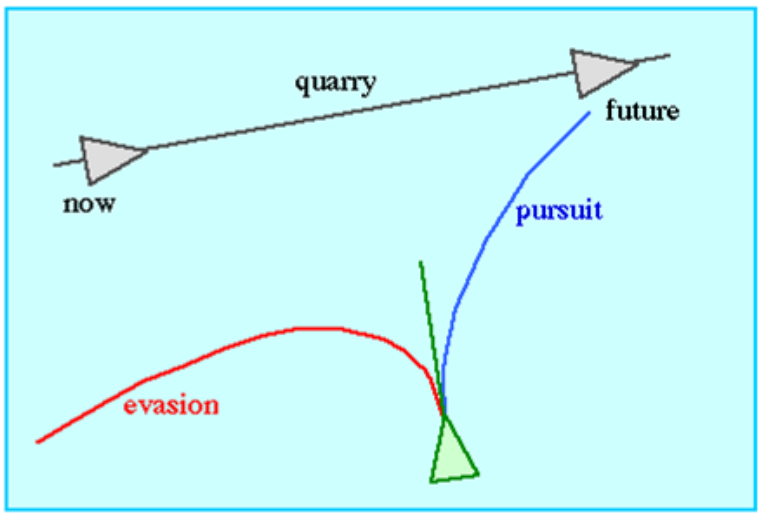
Wobei *gew_geschw* die gewünschte Geschwindigkeit bedeutet und *normalize* eine Funktion ist, die den Grad der Einlenkung zum Ziel bestimmt. Die Variable *steering* würde dann der gewünschte Bewegungs- bzw. Geschwindigkeitsvektor sein, dem wir letztendlich auch folgen wollen.

Für Flee ist das Verfahren ähnlich, jedoch mit einem Flee-Vektor, der vom Ziel wegzeigt.

Eines sollte man jedoch noch beachten: den Zeitpunkt der *Ankunft*. Ab einer bestimmten Entfernung sollte man das Objekt langsamer werden lassen, damit es nicht mit dem Ziel kollidiert oder das Ziel verfehlt *(„Arrival")*. Im Falle des Verfehlens ist ein nett anzusehender, aber bekannter Effekt zu sehen: der Ping-Pong-Effekt.

6.4.2 Verfolgen und Ausweichen („Pursuit and Evasion")

Hier handelt es sich um etwas Ähnliches, welches jedoch etwas komplizierter ist. Die Methode Suchen und Fliehen funktioniert nur für statische Ziele, welches aber in den meisten Anwendungen nicht der Fall ist.



In vielen Spielen findet diese Methode ihre Anwendung. Seien es Ego-Shooter oder Autorennspiele, fast überall wird man von einem Computergegner verfolgt oder man muss diesen verfolgen.

Betrachten wir wieder die obige Grafik, um an ihr den Algorithmus zur Verfolgung zu erklären.

Das grüne Dreieck soll unseren Agenten darstellen und die grauen das zu verfolgende Ziel. Zuerst sei nur das graue Dreieck mit der Beschriftung *now* relevant. Stellen wir uns vor, dass unser Agent direkt auf dieses Objekt zusteuern würde, dann würde es bei gleicher Geschwindigkeit zu einer Endlos-Jagd kommen. Um dieses zu verhindern, versuchen wir eine zukünftige Position und das Zeitintervall T bis zum vermuteten Zusammentreffen mit dem Ziel möglichst exakt zu bestimmen. Aus diesen Daten können wir letztendlich den anzusteuernden Kurs bestimmen.

Eine einfache Abschätzung liefert hier

$$T = D \cdot c$$

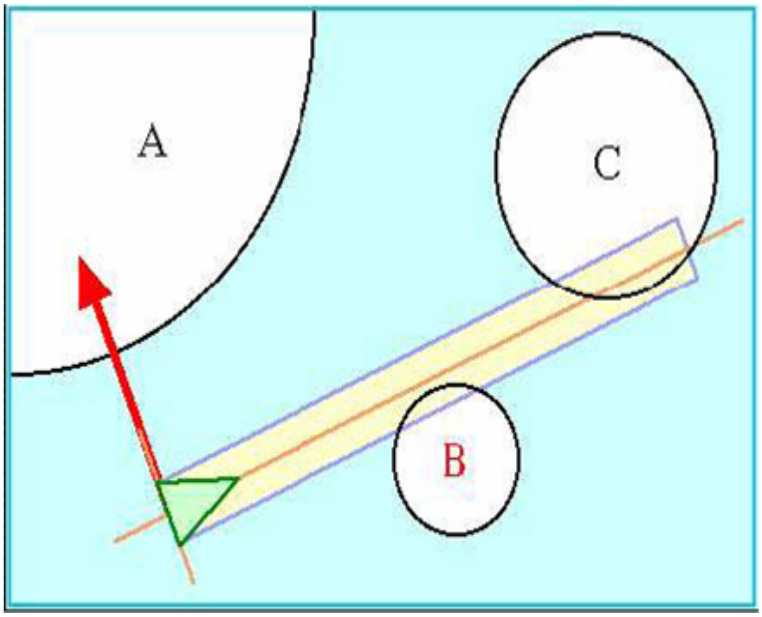
mit D als aktuelle Distanz und c als Drehungsausgleichsparameter [1]

Das Entkommen funktioniert natürlich analog, nur nimmt man den negativen Bewegungsvektor, den man sich durch die Position des zukünftigen Ziels errechnet hat.

Eine kleine Ergänzung zu der Methode des Verfolgens wäre noch, das Ziel neben das zu verfolgende Objekt zu legen, damit man einen Effekt des Begleitens bekommt.

6.4.3 Hindernissen ausweichen („obstacle avoidance")

Ein großes Problem wird jedoch noch durch Hindernisse bereitet. Wie am Anfang mit unserem Strecken-Beispiel beschrieben gibt es hier viele Probleme. Zum Beispiel wird das Hindernis nicht erkannt, da man nicht immer jedes Hindernis einzeln in den Programmtext implementieren kann. Folglich hängt der Agent am Hindernis fest und kommt nicht weiter. Folgende Algorithmen sollen hier Abhilfe schaffen:



Wie auf der Grafik zu sehen, versucht man einen zylinderförmigen Korridor aufzuspannen. Ziel ist es nun, diesen Korridor vor sich freizuhalten. Die Größe des Korridors machen wir je nach Belieben von Geschwindigkeit, Beweglichkeit und Sichtweite des Agenten abhängig.

Nun haben wir unsere Hindernisse A, B und C, wo wir je nach Größe eine so genannte *bounding sphere* aufspannen. Dieses ist eine Zone (hier kreisförmig), in der bei Erkennung Alarm geschlagen wird. Man spannt diese Zone auf, um noch ausreichend Platz zum Ausweichen zu haben. Bei zu kleinen Zonen ist unser Agent nicht wendig genug, um diese zu umgehen, ergo kommt es wieder zum Zusammenstoß.

Die Funktion sieht in etwa so aus:

Falls

$$d(zentrum\_C, zentrum\_B) < Radius\_C + Radius\_B$$

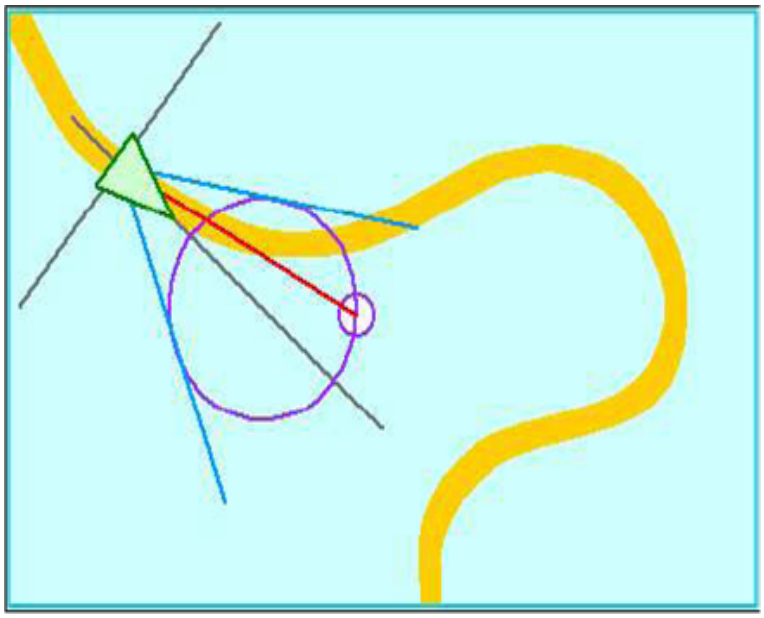
weiche aus (d = Abstand).

Betrachtet wird dieses auf dem Koordinatensystem ausgehend vom Agenten, und erfasst werden nur jene Hindernisse, deren *bounding sphere* in Reichweite des Korridors ist. Eine oft verbreitete Lösung des Ausweichens ist es, hier den invertierten Offsetvektor als neuen Steuerungsvektor zurückzugeben (roter Pfeil).

Wie man feststellt, kann man mit dieser Methode schnell und meist auch effektiv den Hindernissen ausweichen. Von der Rechenintensität gehört dieses auf jeden Fall zu den besten Methoden. Außerdem kann man diese Methode auch auf ein Feld von sich asynchron bewegenden Agenten anwenden, die sich gegenseitig ausweichen sollen.

6.4.4 *Wandern*

Das so genannte *Wandern* versucht etwas mehr Realismus auf unseren Agenten einwirken zu lassen. Kein Mensch geht eine lineare Strecke und es sieht natürlich auch unrealistisch aus, wenn unsere Agenten dieses tun. Das *Wandern* ist eine einfache Methode, um dieses zu verhindern:

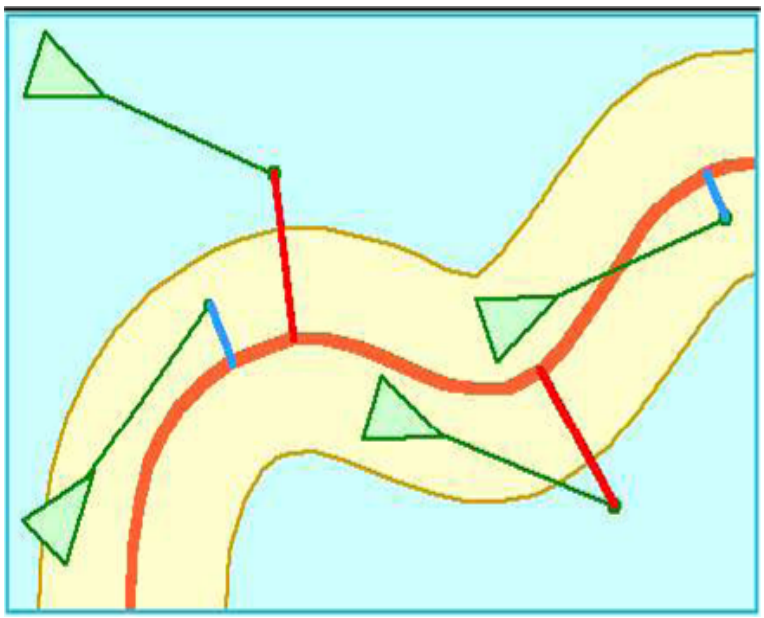


Um ein paar zufällige Bewegungen in unseren sich bewegenden Agenten einzubauen, legen wir vor diesen in einem beliebigen Abstand einen Kreis. Dieser Abstand ist genauso wie die Größe des Kreises abhängig von der Intensität der abweichenden Bewegung, die unser Agent bekommen soll. Unseren Bewegungsvektor legen wir nun auf den Kreis (roter Strich) und lassen ihn sich zufällig auf diesem bewegen. Wichtig ist aber, dass die Bewegung gleichmäßig bleibt, damit keine ,,Zuckungen" entstehen.

Und schon haben wir mit einfachen Mitteln eine fehlerhaft aussehende Bewegung erstellt - oder wie Sisters of Mercy (Band) es so schön sagen: ,,Some Girls Wander by Mistake".

6.4.5 Weg weisend *(,,path following")*

Kommen wir zu etwas anderem. Um die meisten Dinge haben wir uns bereits gekümmert, jedoch nicht darum, dass unser Agent eine bestimmte Strecke gehen soll. Dieses bewirkt man mit *path following* ! Zum Beispiel, wenn eine Spielfigur in einem Computerspiel einen Gang entlang gehen soll. Man könnte mehrere Koordinaten als Wegpunkte angeben, wo unser Agent entlanggehen soll. Dieses würde jedoch bei großen Strecken zu unflexibel sein. Also versucht man, mit den eh schon definierten Wänden bzw. Begrenzungen klarzukommen:



Die Idee ist es, die Begrenzungen (Wände) zu betrachten und einen Weg zwischen diesen (oranger, dicker Strich) zu ermitteln. Der ermittelte Weg zwischen den Begrenzungen gilt als optimaler Pfad, und man versucht sich an diesem zu orientieren. Nennen wir den Abstand zwischen dem optimalen Pfad und der Wand ,,b". Als zweites versuchen wir linear vorherzusagen, wo unser Agent hinläuft indem wir einen Punkt ,,A" in gewissem Abstand vor dem Agenten bestimmen. Mittels Punkt ,,A" und seinem Abstand ,,c" zum optimalen Pfad können wir folgendes aufstellen:

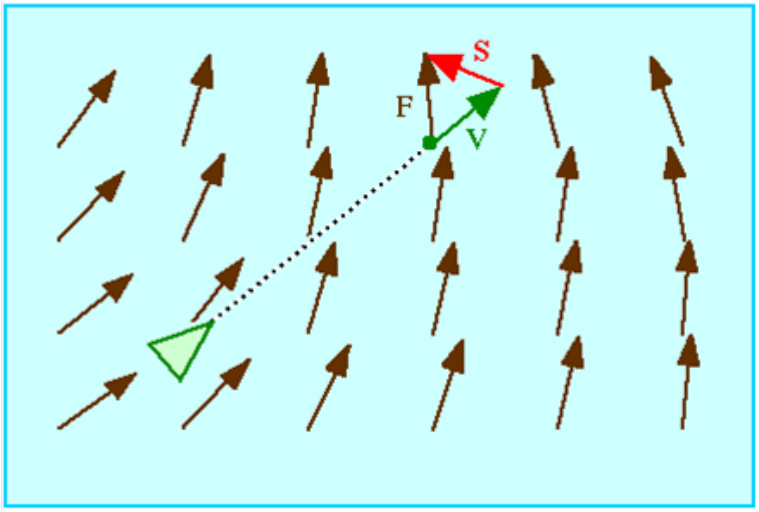
Falls

$$c > b$$

lenke zum optimalen Pfad ein.

6.4.6 Bewegung durch ein Vektorfeld

Für Flussläufe oder ähnliches kann man folgenden Algorithmus verwenden:



Mit Hilfe eines ganzen Feldes, wo jeder Punkt einem Vektor zugeordnet ist (kurz Vektorfeld), kann man einen sich frei bewegenden Agenten hindurchschicken, der sich an diesen Vektoren orientiert. Ein Anwendungsgebiet wäre zum Beispiel ein Schiff, welches einer Strömung eines Flußlaufes ausgeliefert ist. Das Schiff bringt seine Eigenbewegung durch Vektor V mit (grüner Pfeil). Wir bestimmen uns wieder einen zukünftigen Punkt, wo wir mit dem Punkt zugehörigen Vektor F den Steuerungsvektor S durch einfache Subtraktion bestimmen können. Dieser Vektor S wirkt also auf den momentanen Bewegungsvektor V ein - und unser Schiff/Agent wird durch diesen auf einen neuen Kurs gebracht.

6.5 Exkurs

6.5.1 Schwärme *(„Flocking")*

Zuletzt noch ein kleiner, aber interessanter Exkurs in den Bereich *Flocking*. *Flocking* bedeutet, dass mehrere Agenten miteinander agieren. Was passiert zum Beispiel, wenn mehrere Agenten einem Objekt folgen oder einen Pfad entlanggehen. In den meisten Fällen würden diese kollidieren, auch wenn man den oben beschriebenen *Hindernis ausweichen* -Algorithmus benutzen würde (abgesehen davon, dass dieser auf mehrere Agenten angewandt sehr konfus aussieht). Deshalb hat man sich eine Idee dem Magnetfeld abgeguckt: Zwischen den Agenten wird sowohl ein Abstoßungsvektor und ein Anziehungsvektor aufgestellt. Diese beiden Vektoren wirken wieder auf den Bewegungsvektor ein. Der Abstand zwischen den Agenten pendelt sich durch die beiden Vektoren irgendwann ein, und auch nach Veränderungen in diesem Feld werden die Agenten sich wieder auf einen bestimmten Abstand einpendeln. Mehr hierzu finden Sie unter [2.]

6.6 Literaturverzeichnis

1. Craig W. Reynolds: Steering Behaviours For Autonomous Characters, Sony Computer Entertainment, 1999

2. Craig W. Reynolds: Flocks, Herds and Schools: A Distributed Behavioral Model, Symbolics Graphics Division, 1987

3. Stuart J. Russel and Peter Norvig (1995) Artificial Intelligence, Prentice-Hall

4. http://www.red3d.com/cwr/steer/

5. http://opensteer.sourceforge.net/

Kapitel 7

Virtual Reality Modeling Language (VRML)

7.1 Einleitung

„Second Life“heißt das Stichwort, aus seinem tristen Alltagsleben zu entkommen.[1] Bereits über 240.000 Menschen auf der ganzen Welt haben sich dieser Online-Welt angeschlossen, um sich zumindest in ihrem „zweiten Leben“langersehnte Wünsche erfüllen zu können. Wie im richtigen Leben wird in dieser Welt eingekauft, in einem Haus gewohnt, sich verliebt oder sogar geheiratet.

Nichtsdestotrotz handelt es sich hier nicht um das richtige Leben, sondern nur um eine virtuelle Realität im Internet. Eine virtuelle Realität ist eine in Echtzeit generierte Computersimulation, in die ein Benutzer eintauchen kann. Im Allgemeinen wird dazu nicht nur spezielle Software benötigt, sondern auch ein Head-Mounted-Display, über welches der Benutzer in die Welt eintauchen kann. Zudem werden oftmals auch Tracking-Systeme wie Datenhandschuh oder -anzug gebraucht, durch welche die Position und die Handlung des Anwenders bestimmt werden kann.

In diesen Zusammenhang gehört auch die 3D-Beschreibungssprache VRML (Virtual Reality Modeling Language), die es ermöglicht, die 3-dimensionale Darstellung von Objekten oder Welten zu beschreiben. Hier wird allerdings weder ein Head-Mounted-Display noch Tracking-Systeme benötigt, lediglich ein VRML-Browser, der die Welt darstellt. VRML ähnelt der Web-Sprache HTML, da sie ebenso dateiorientiert und ein Textformat ist und im Quellcode ebenso Schlüsselwörter auftauchen, die für die Beschreibung der Szene zuständig sind.[2] Desweiteren ist VRML plattformunabhängig, somit auf jedem Computer nutzbar, und vollständig freigegeben sowie der Öffentlichkeit zugänglich gemacht.[3] Zu den weiteren Eigenschaften gehören, dass mit Hilfe von VRML sowohl statische als auch dynamische Welten, in welcher der Benutzer interagieren kann, erzeugt werden können. Die VRML-Datei hat die Endung *.wrl, was für world

[1] siehe www.secondlife.com/whatis/.

[2] siehe www.uni-kassel.de/hrz/anwendungen/matthias/hrz-info/vrml/vrml.html.

[3] siehe O. Schlüter: VRML. Sprachmerkmale, Anwendungen, Perspektiven, Köln 1998, S. 7 f.

steht. VRML findet in vielen Gebieten Anwendung. Dazu gehören u.a. die naturwissenschaftlichen Bereiche, wo es zur Erzeugung von dreidimensionalen Modellen benötigt wird, oder auch die Industrie sowie das Marketing, wo Objekte oder Firmen virtuell betrachtet bzw. besichtigt werden können.

Im Folgenden soll zunächst ein Einblick in den geschichtlichen Hintergrund von gewährt werden. Dann sollen sowohl die technischen als auch die sprachlichen Grundlagen kurz vorgestellt werden, um schließlich in Hinblick auf die aktuelle Situation zu analysieren, warum VRML sich nicht auf dem Markt für 3D-Webtechnologien behaupten konnte.

7.2 Geschichtliche Entwicklung

In den 90er Jahren wuchs die Akzeptanz für das Internet stark durch die Einführung des World Wide Webs an, das durch sein benutzerfreundliches und mediales Aussehen auch Computerlaien einen leichten Umgang mit dem Internet ermöglichte.[4] Dadurch wuchs auch immer stärker der Wunsch, die virtuelle Realität ins World Wide Web zu integrieren. Dazu allerdings musste man eine Sprache entwickeln, die konzeptionell der Web-Sprache HTML entsprach. Das heißt, sie musste es ermöglichen, 3D-Welten ähnlich wie HTML-Dokumente speichern zu können.

Mai 1994 stellten Mark Pesce und Tony Parisi schließlich einen Prototyp namens „Labyrinth“eines 3D-Viewers integriert in den Browser Mosaic bei der ersten WWW-Konferenz in Genf vor.[5] Angelehnt an HTML bezeichnete man die Sprache, die die 3-dimensionale Darstellung im WWW ermöglichen sollte, als Virtual Reality Markup Language[6], wobei Markup später durch Modeling ersetzt wurde, da dies mehr ihrer Bedeutung entsprach.

Pesce und Parisi eröffneten eine Mailing-Liste zum Thema VRML, um allen Interessenten[7] die Möglichkeit zu bieten, über Eigenschaften, die diese neue Sprache aufweisen sollte, zu diskutieren und somit an der Entwicklung mitwirken zu können.[8] Bereits im Oktober 94 konnte ein Entwurf auf der 2. WWW-Konferenz vorgestellt werden. Im April 95 wurde schließlich die erste offizielle Spezifikation VRML 1.0 veröffentlicht. Dadurch konnten zunächst nur einfache, statische 3D-Welten beschrieben und durch einen Browser dargestellt werden, in welchen der Benutzer sich zwar bewegen, aber noch nicht interagieren konnte. Bei der Entwicklung wurde bewusst darauf geachtet, sich auf das Wichtigste zu beschränken, um in kurzer Zeit einen hohen Qualitätstandard bei Browsern und Entwicklertools zu ermöglichen.[9] Aufgrund der schnellen Weiterentwicklung des Internets musste die Entstehung von VRML schnell voranschreiten, so dass man auf bereits existierendem aufbaute.[10] Aus diesem Grund stammen die

[4] siehe S. Matsuba und B. Roehl: VRML. Das Kompendium, München 1996, S. 149 f.
[5] siehe www.playfulworld.com/career.html.
[6] kurz VRML.
[7] sowohl Privatpersonen als auch Institutionen wie SGI, Netscape oder Apple.
[8] siehe O. Schlüter, a.a.O., S. 15.
[9] siehe ebda., S. 17.
[10] siehe K. Zeppenfeld: Lehrbuch der Grafikprogrammierung. Grundlagen, Programmierung, Anwendung, München 2004, S.366.

3D-Eigenschaften des Dateiformats *.wrl größtenteils von Open Inventor[11]. Es wurde zusätzlich mit einer Komponente zur Kompatibilität mit dem Internet ausgestattet.

Die Veröffentlichung eines gemeinsamen Standards ermöglichte zudem, dass eine virtuelle Welt, welche durch VRML 1.0 beschrieben wurde, nicht nur auf einem bestimmten Browser läuft, sondern auf allen, die VRML 1.0 konform sind. Dies führte zu einer höheren Sicherheit bei Entwicklern von Browsern und Welten.

Durch die Gründung der VRML Architecture Group[12] im Jahr 94 konnten alle Aktivitäten, die sich um die Entwicklung von VRML drehten, gesammelt und koordiniert werden, so dass man einen gemeinsamen Standard schaffen konnte. Ende 95 wurde schließlich auf der VRML-Konferenz in San Diego die Idee einer Weiterentwicklung zu VRML 2.0 vorgeschlagen.[13] Anfang des nächsten Jahres wurde ein Request for Proposals von der VAG aufgerufen, um Alternativen zur Moving-World-Eigenschaft für die neue Spezifikation zu sammeln und auszudiskutieren. Man entschied sich daraufhin für das Moving-World-Proposal, so dass im August 96 auf der SIGGRAPH-Konferenz endgültig die neue Spezifikation VRML 2.0 vorgestellt und freigegeben werden konnte. Diese ermöglichte animierte Welten zu beschreiben, in welchen der Benutzer zudem interagieren konnte, insbesondere da nun Skriptsprachen[14] in den Quellcode integriert werden konnten. Außerdem waren zwar Multi-User-Welten bereits machbar, aber durch die Spezifikation noch nicht erlaubt.

Nach der Veröffentlichung von VRML 2.0 wurde das VRML-Konsortium[15] gegründet, welchem sich über 35 Firmen und Organisationen[16] anschlossen. Es hatte die Aufgabe die Weiterentwicklung von VRML voranzubringen und zu koordinieren. Dazu bestand es aus einem Direktorat und einzelnen technischen Arbeitsgruppen[17], welche zu diversen Gebieten Vorschläge und Konzepte zur Erweiterung von VRML erarbeiten mussten, sowie einem VRML-Reviewer-Board[18], das die Vorschläge der WG kontrollieren musste.

Im April 97 wurde die Spezifikation VRML 2.0 unter dem Namen VRML 97 zum Draft International Standard 14772 der International Standard Organization erklärt, welche im September schließlich endgültig als ISO-Standard galt. Dies führte zur internationalen Anerkennung und Akzeptanz unter den Firmen und Privatpersonen.

In den folgenden Jahren versuchte man, die Entwicklung von VRML weiter voranzutreiben. Im Jahr 99 diskutierte man über die Entwicklung einer weiteren Spezifikation namens VRML NG[19] bzw. VRML 99 auf der vierten internationalen Konferenz über 3D Webtechnologien.[20] Allerdings kam es nie zur Fertigstellung. Die Entwicklungen wurden abgebrochen und man

[11] Dateiformat der Firma Silicon Graphics zur Erstellung von 3D-Szenen.
[12] kurz VAG.
[13] siehe O. Schlüter, a.a.O., S. 18 f.
[14] z.B. JavaScript.
[15] VRMLC.
[16] u.a. SGI, Netscape, SUN und Microsoft.
[17] Working Groups, kurz WG.
[18] kurz VRB.
[19] Next Generation.
[20] siehe http://de.wikipedia.org/wiki/vrml.

begann, sich auf den Nachfolger X3D zu konzentrieren. Dieser allerdings konnte sich bisher noch nicht gegen die Großen der Webtechnologien[21] durchsetzen. Was VRML betrifft, verschwand mit dem Ende der Weiterentwicklung auch das Interesse und die Popularität.

7.3 Browser und PlugIns

VRML-Browser sind eine notwendige Software, um die durch VRML beschriebenen virtuellen Welten darstellen zu können. Das heißt, sie sind der Interpreter von VRML. Als eigenständige Anwendungen besitzen sie nicht die Möglichkeit, HTML-Seiten darzustellen.[22] Um die wrl-Dateien in einem bekannten WWW-Browser[23] anzeigen lassen zu können, muss ein PlugIn installiert werden.

Beim Starten einer wrl-Datei wird zunächst der Quellcode interpretiert.[24] Dazu werden die Datenstrukturen, die zur Darstellung der Szene benötigt werden, im Speicher abgelegt.[25] Nach den Berechnungen wird gerendert und somit die Welt auf den Bildschirm projiziert. Aus diesem Grund ist die Rendering Engine auch der wichtigste Bestandteil des Browsers, die zudem auch die meiste Rechenleistung benötigt. Insbesondere wird sie bei jeder Bewegung des Anwenders durch die Welt aufgerufen, um die Bilder auf dem Bildschirm zu aktualisieren. Das heißt, der Browser generiert die Szene in Echtzeit, weshalb der Entwickler selbst bei der Programmierung in VRML den späteren Betrachterstandpunkt nicht mitberücksichtigen muss.

Inzwischen gibt es diverse Browser bzw. PlugIns, die zwar funktionell keine großen Unterschiede aufweisen, aber zum einen in Bezug auf die Konformität mit dem Standard und zum anderen im Aussehen ihrer Werkzeugleisten voneinander differenzieren können.

Im Allgemeinen weisen alle zwei Werkzeugleisten auf, die folgende Auswahlmöglichkeiten bieten:[26]

- Horizontal: Veränderung der Position des Anwenders innerhalb der Welt
    1. view (Auswahl verschiedener durch den Entwickler ausgewählter Betrachtungsstandpunkte)
    2. align (Wiederherstellung der Ausgangsstellung)
    3. restore (Wiederherstellung der Ausgangsszene)
    4. fit (Anpassung der Größe der Welt auf die Größe des Bildschirms)
- Vertikal: Art der Navigation innerhalb der Welt

[21] z.B. Adobe Flash.
[22] siehe K. Zeppenfeld, a.a.O., S. 191.
[23] z.B. Mozilla Firefox, Internet Explorer.
[24] siehe S. Matsuba und B. Roehl, a.a.O., S. 192.
[25] siehe ebda., S. 158 f.
[26] siehe K. Zeppenfeld, a.a.O., S. 368.

- walk, fly, study
  1. plan (in horizontaler Ebene bewegen)
  2. pan (in vertikaler Ebene bewegen)
  3. turn (in horizontaler Ebene drehen)
  4. roll (in vertikaler Ebene drehen)

7.4 VRML-Syntax

Eine VRML-Datei besteht aus zwei wesentlichen Elementen: Dem Header, der immer am Anfang stehen muss, damit der Browser die Datei auch erkennt, und einer Aufzählung der benötigten Objekte in hierarchischer Anordnung.[27] Der Header hat folgenden Aufbau:[28]

#VRML, verwendete VRML-Spezifikation, Zeichensatz des Quellcodes[29]

Ansonsten leitet das #-Zeichen einen einzeiligen Kommentar ein.

Ein wichtiger Bestandteil der VRML-Syntax sind die Knoten, die durch einen Typ-Namen[30] und ein Paar geschweifter Klammern gekennzeichnet sind.[31] In der Spezifikation gibt es eine Reihe von vordefinierten Knoten, die dazu benötigt werden, einzelne Objekte in der Welt zu beschreiben. Zudem besteht die Möglichkeit, einem Knoten durch das Schlüsselwort DEF einen beliebigen Namen[32] zuzuteilen, so dass man ihn bei Bedarf durch „USE + zugeteilter Name“mehrmals verwenden kann, ohne die Eigenschaften erneut aufzählen zu müssen. Desweiteren unterscheidet man hauptsächlich zwei Arten — zum einen gibt es die Gruppenknoten, die mehrere Knoten zu einer Einheit gruppieren, zum anderen die Blattknoten, die dies nicht tun.[33] Hingegen werden Knoten, die keine Objekte beschreiben, sondern nur für bestimmte Einstellungen des Browsers sorgen, als unabhängig bezeichnet.[34]

Die geschweiften Klammern von Knoten können leer sein oder in beliebiger Reihenfolge als Felder bezeichnete Attribute beinhalten, die die Eigenschaften[35] des Objekts bestimmen. In der Spezifikation ist festgelegt, welche Felder von welchem Datentyp zu einem Knoten gehören sowie welchen Wert diese standardmäßig haben.[36] Allerdings taucht der Datentyp nicht im Quellcode auf. Zudem muss man nur die Felder aufzählen, für die ein anderer Wert als standardmäßig vordefiniert vorgesehen ist. Im Allgemeinen unterscheidet man, ob ein Attribut nur einen Wert

[27] siehe O. Schlüter, a.a.O., S. 35.
[28] siehe K. Zeppenfeld, a.a.O., S. 371.
[29] z.B. #VRML V2.0 utf8 oder #VRML V1.0 ascii.
[30] beginnt immer mit Großbuchstaben.
[31] siehe K. Zeppenfeld, a.a.O., S. 369 f.
[32] muss mit einem Großbuchstaben beginnen.
[33] siehe O. Schlüter, a.a.O., S. 44.
[34] siehe ebda., S. 63.
[35] z.B. Größe, Farbe, Position.
[36] siehe O. Schlüter, a.a.O., S. 40 f.

annimmt, was als singlevalued bezeichnet wird, oder mehrere (multivalued), wo die einzelnen durch Kommata oder Leerzeichen getrennt zwischen zwei eckigen Klammern stehen.[37] Desweiteren kann es sich beim Feldwert auch um einen weiteren Knoten handeln. Im Gegensatz zu den Knoten werden die Typ-Namen von Attributen kleingeschrieben, außer bei zusammengesetzten Bezeichnungen, wo ab dem 2. Bezeichner jeder groß beginnt. Daraus ist erkenntlich, dass VRML zwischen Groß- und Kleinschreibung unterscheidet.

Zu den wichtigsten Knoten gehört Shape{} mit den Feldern appearance und geometry, da dieser ein neues Objekt definiert.

Ab VRML 2.0 besteht zusätzlich die Möglichkeit eigene Knoten durch das Schlüsselwort PROTO zu definieren, so dass man nicht mehr auf die in der Spezifikation enthaltenen beschränkt ist.[38]

7.5 Grundlagen von VRML

In der Spezifikation sind bereits die vier grundlegendsten Körper vordefiniert, aus denen sich auch weitere komplexere Objekte bilden lassen. Das bietet den Vorteil, dass diese nicht mehr eigenhändig aus Polygonen aufgebaut werden müssen, was zum einen dem Entwickler viel Arbeit erspart und zum anderen die Komplexität verringert, da die Körper optimal aus Polygonen zusammengesetzt sind.[39]

Zu den VRML-Primitiven gehören:[40]

Körper	**Kugel**	**Quader**	**Zylinder**	**Kegel**
Knoten	Sphere{}	Box{}	Cylinder{}	Cone{}
Felder	radius 1.0	size 2.0 2.0 2.0	radius 1.0 height 2.0 bottom TRUE side TRUE top TRUE	bottomRadius 1.0 height 2.0 side TRUE bottom TRUE
Bild				

Diese Grundkörper sind u.a. mögliche Werte vom Feld geometry.

[37] siehe K. Zeppenfeld, a.a.O., S. 370 f.
[38] siehe O. Schlüter, a.a.O., S. 35.
[39] siehe www.uni-kassel.de/hrz/anwendungen/matthias/hrz-info/vrml/vrml.html.
[40] bei Felder sind die Standardwerte angegeben.

Mit Hilfe eines Koordinatenknotens lassen sich Punkte beschreiben, die für die Darstellung von Linien essentiell sind. Dementsprechend hat dieser ein Attribut namens point, dessen Werte die x-, y-, z- Koordinate angeben.[41] Eine Punktemenge kann nun folgendermaßen beschrieben werden:

```
Shape{
   appearance Appearance{
     material Material{}
   }

   geometry PointSet{
     coord Coordinate{
       point[
         0 0 0.5 # erster Punkt
         1 5.25 1 # zweiter Punkt
         2 0 2 # dritter Punkt
       ]
}  } }
```

Desweiteren wird jedem Punkt ein Koordinatenindex beginnend bei 0 zugeordnet, um Linien ohne die wiederholte Angabe der Raumkoordinaten der einzelnen Punkte erstellen zu können. Folgender Code zeigt, wie sich Linien bilden lassen:

```
geometry IndexedLineSet{
  coord Coordinate{
    point[
      0 0 0.5
      1 5.25 1
      2 0 2
      1 1 3
     ]
    coordIndex[
      0 1 2 -1 # durch -1 werden mehrere Linien voneinander getrennt
      2 3 # neue Linie
    ]
} }
```

Schließlich lassen sich durch Ersetzen von IndexedLineSet in IndexedFaceSet Flächen erzeugen, wobei das zusätzliche boolsche Attribut solid angibt, ob durch die Fläche durchgesehen werden kann.

[41] rechtshändiges Koordinatensystem: x-Achse horizontal, y-Achse vertikal, z-Achse vom Bildschirm auf den Betrachter. siehe K. Zeppenfeld, a.a.O., S. 373 ff.

Eine weitere grundlegende Eigenschaft von Objekten ist ihre Erscheinung, die durch den Knoten Appearance bestimmt wird. Als Felder stehen der Materialknoten zur Auswahl, der die allgemeine Oberflächenbeschaffenheit angibt, oder der Texturknoten, der ganze Bilder bzw. Videoclips auf das Objekt projiziert.

Zu den wichtigsten Feldern des Materialknotens gehören diffuseColor, der die Farbe des Objekts durch normierte RGB-Werte beschreibt, und transparency, wodurch die Durchsichtigkeit des Objekts mit Annäherung an den Wert 1 steigt.

```
appearance Appearance{
   material Material{
      diffuseColor 0 0 1 # blaues Objekt
      transparency 0.75 # fast vollst"andig durchsichtig
}  }
```

Zur Projektion eines gesamten Bildes[42] auf ein Objekt wird der ImageTexture-Knoten benötigt. Befinden sich VRML-Datei und Bild am gleichen Speicherort, muss für das Feld url nur der Name der Datei zwischen zwei Anführungsstriche gesetzt werden, ansonsten der relative Pfad.

```
appearance Appearance{
   texture ImageTexture{
      url "mauer"
}  }
```

Auf gleiche Weise können durch den MovieTexture-Knoten mpeg-Videos auf ein Objekt projiziert werden.

Zuletzt sind noch Transformationen eine wichtige Grundlage, da Objekte beispielsweise grundsätzlich immer in den Ursprung des Koordinatensystems gesetzt werden.[43] Der Transformationsknoten gehört zu den Gruppenknoten, weshalb in seinem Feld children zwischen den eckigen Klammern sämtliche Objekte stehen, für die die gleichen Veränderungen ausgeführt werden sollen.

Zur Auswahl stehen folgende Modifikationen:

```
Transform{
   children[
       Shape{
          appearance Appearance{
```

[42] gif-, jpg- oder png-Dateien.
[43] siehe K. Zeppenfeld, a.a.O., S. 382.

```
            material Material{
                diffuseColor 0.3 0 0
            }
        }

        geometry Box{
            size 4.0 1.0 2.0
        }
    }
]
translation 0.15 1.0 0.0 # um 0.15 in x- und um 1 in y-Richtung
                         # verschoben
rotation 0 0 1 0.75 # Drehung: Rotationsachse durch den Ursprung
                    # und (0, 0, 1), Drehwinkel 0.75 Radiant
scale 1 0 -2 # Vergr"o"serung um 1 in x-Richtung
             # Verkleinerung um 2 in z-Richtung
}
```

Hierbei muss man bei der Rotation beachten, dass ein positiver Drehwinkel eine Drehung gegen den Uhrzeigersinn bedeutet. Zudem wird ein Objekt immer relativ zum Ursprung des Koordinatensystems vergrößert bzw. verkleinert. Aus diesem Grund interpretiert der Browser unabhängig vom Auftreten im Transformationsknoten immer zuerst die Skalierung, dann die Rotation und zuletzt die Verschiebung. Dies kann unterbunden werden, indem für jede Transformation ein eigener Knoten vorgesehen wird, so dass durch Verschachtelung der einzelnen eine andere Reihenfolge erzwungen werden kann.

7.6 Der Szenengraph

Die hierarchische Struktur der Objekte innerhalb einer VRML-Datei wird als Szenengraph bezeichnet und stellt einen Baum dar.[44] Hierbei ist die Wurzel die gesamte virtuelle Welt.[45] Blattknoten haben entsprechend der Definition eines Baums keine Kindknoten und unterscheiden sich somit eindeutig von den Gruppenknoten, welche mehrere Objekte in bestimmten Eigenschaften zusammenfassen. Somit sorgen die Gruppenknoten für die hierarchische Struktur, die den Szenengraphen ausmacht, da Knoten anderen Knoten durch diese untergeordnet werden.[46]

[44] siehe K. Zeppenfeld, a.a.O., S. 383.
[45] siehe ebda., S. 322.
[46] siehe O. Schlüter, a.a.O., S. 35.

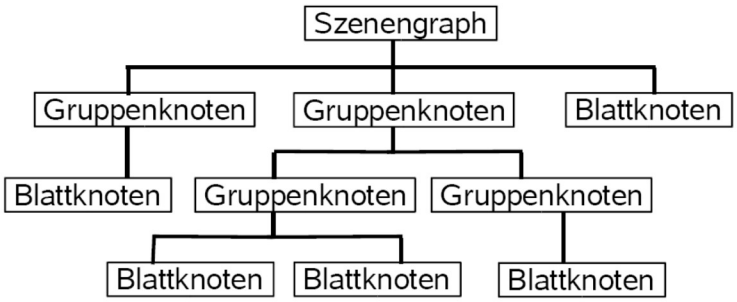


Abbildung 7.1: allgemeines Beispiel für einen Szenengraph

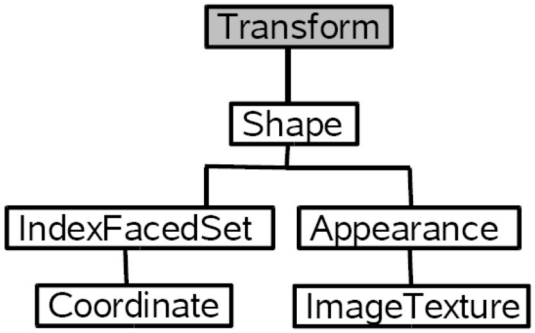


Abbildung 7.2: Beispiel für einen Szenengraph

7.7 Überblick zu dynamische Welten

Das Moving-World-Proposal ist eines der wichtigsten Erneuerungen von VRML 2.0 gegenüber VRML 1.0. Knoten können jetzt zusätzlich zu den Feldern Ereigniseingänge (eventIn) und -ausgänge (eventOut) besitzen, über welche sie die Meldung über Statusveränderung anderer Knoten erhalten bzw. über eigene Veränderungen anderen berichten können.[47] Das heißt, Ereignisse sind ein Mittel, über welches Knoten miteinander kommunizieren können. Desweiteren können auch Felder vom Typ exposedField ein eventIn und eventOut besitzen, das direkt mit dem Wert des Feldes verbunden ist, so dass dieser durch andere Knoten verändert werden kann. Ein solches Feld bezeichnet man auch als public, wohingegen diejenigen, dessen Wert zur Laufzeit nicht mehr verändert werden kann, private sind.[48]

Der Pfad, über welchen die Ereignisse versendet werden, bezeichnet man als Route. Diese kann immer nur vom eventOut des einen Knotens zum eventIn des anderen gebildet werden.[49] Eine Route ist nur gültig, wenn zum einen beide Knoten über DEF Namen zugeordnet wurden und

[47] siehe K. Zeppenfeld, a.a.O., S. 388 f.
[48] siehe O. Schlüter, a.a.O., S. 42.
[49] siehe ebda., S. 72.

zum anderen die Datentypen des Ereignisausgangs des einen Knotens und Ereigniseingangs des anderen übereinstimmen. Darüberhinaus sind Routen selbst keine Knoten. Im Allgemeinen haben sie folgenden syntaktischen Aufbau:

ROUTE knotenname1.feldName1_changed TO knotenname2.set_feldName2

Das Ausführungsmodell von VRML beschreibt, wie Routen und Ereignisse interpretiert werden.[50] Der Browser überprüft hierzu bei jedem Taktsignal, ob ein eventOut ein Ereignis erzeugt hat und daraufhin, ob auch eine Route für diesen Ausgang vorhanden ist. Knoten können dann auf den Erhalt einer Meldung reagieren, indem sie selbst Ereignisse versenden. Diesen Vorgang bezeichnet man auch als Ereignis-Kaskade.

Ereignisse und Routen sind zwei der wichtigsten Voraussetzungen für Animationen. Darüberhinaus benötigt man einen Zeitknoten (TimeSensor), der die zeitlichen Abläufe steuert. Zu den Attributen gehören startTime und stopTime, cycleInterval, das die Dauer eines Zeitintervalls in Sekunden angibt, sowie loop, das durch Setzen von TRUE Endlosschleifen erzeugt.[51] Animationen, die durch VRML beschrieben werden, sind von einfacher Natur, weshalb nur wenige Zustände für die Animation angegeben werden müssen. Die notwendigen Zwischenzustände werden vom Browser mittels Interpolatorknoten berechnet, was nicht nur dem Entwickler Arbeit erspart, sondern auch die Performanz steigert. Dazu werden im Feld key alle Zeitpunkte angegeben, zu welchen die im Feld keyValue angegebenen Werte angenommen werden sollen. Die Attribute set_fraction und value_changed sind der Ereigniseingang und -ausgang. Bei den Interpolatoren unterscheidet man diverse Typen, die zur Berechnung einer speziellen Animation zuständig sind. Zu den wichtigsten gehört der PositionInterpolator, der das Bewegen von Objekten von einem Ort zum anderen durch Translationen ermöglicht. Komplexere Animationen können durch Integration einer Skriptsprache ermöglicht werden.

Eine weitere Erneuerung bei VRML 2.0 stellen die Interaktionen dar, die es dem Anwender ermöglichen, ins Geschehen der Welt einzugreifen. Hierzu benötigt man Sensoren, die bestimmte Handlungen des Betrachters anderen Knoten mitteilen können.[52] Beispielsweise reagiert der ProximitySensor auf die Annäherung an ein Objekt und der TouchSensor auf Aktionen durch die Maus[53]. Der TimeSensor hingegen gilt als vom Benutzer unabhängig.

7.8 Probleme von VRML

Bereits zu Beginn der Entwicklung von VRML gab es zahlreiche Probleme, die gelöst werden mussten. Beispielsweise war zu diesem Zeitpunkt die Internet-Bandbreite noch sehr niedrig.[54] Diese spielte allerdings eine große Rolle beim Laden von VRML-Welten sowie beim Herunter-

[50] siehe ebda., S. 73.
[51] siehe K. Zeppenfeld, a.a.O., S. 389.
[52] siehe K. Zeppenfeld, a.a.O., S. 391.
[53] z.B. ziehen, klicken, berühren.
[54] siehe O. Schlüter, a.a.O., S. 166 f.

laden der notwendigen Browser-Software. Anfänglich ließ sich das Problem zum Teil dadurch lösen, dass die Welten komprimiert wurden. Browser konnten diese Dateien automatisch entpacken und anschließend laden. Desweiteren versuchte man, Polygon-Reduktions-Algorithmen einzusetzen, um effizienteren VRML-Code zu erzeugen und so die Komplexität zu verringern. Außerdem kann mit Hilfe des LOD-Knotens (Level of Detail) eine Liste für die Darstellung eines Objekts abhängig von der jeweiligen Entfernung des Betrachters erstellt werden, was ebenfalls die Komplexität verringert. Darüber hinaus ist das Problem heutzutage hinfällig, da beispielsweise DSL bereits große Internet-Bandbreiten aufweist.

Ebenso hing damals die Rendering-Geschwindigkeit von der Anzahl der darzustellenden Polygone ab, welche durch ineffizienten Code leicht hohe Maße annehmen konnten. Jedoch hat diese eine große Bedeutung, da bei Bewegung durch die Welt ständig aktualisiert werden muss. Allerdings löste sich das Problem nicht nur durch die verbesserte Leistung des Computers, sondern auch durch die Integration von 3D-Beschleunigern in die meisten Grafikkarten.

Die anfängliche Dynamik in der Entwicklung von VRML hatte auch seine Nachteile, denn die Browser mussten mit VRML konform sein.[55] Allerdings wurden zu Beginn Welten noch auf eine bestimmte Browserversion angefertigt, da nicht alle Browser dasselbe darstellten. Dies führte dazu, dass die Welten oft aktualisiert werden mussten, wenn beispielsweise neue Versionen veröffentlicht wurden. Aus diesem Grund gab es bei Entwicklern von Welten zum Teil wenig Anerkennung. Jedoch änderte sich dies, sobald VRML als ISO-Standard erklärt wurde, da Veränderungen nun nicht mehr so schnell vonstatten gehen konnten aufgrund der zahlreichen Formalitäten für die Aufnahme in den Standard. Diesbezüglich stellen die meisten Browser, die sich als einer Spezifikation konform bezeichnen, dasselbe dar.

Im Allgemeinen ist die zum Teil hohe Komplexität der VRML ein Auslöser für diverse Probleme.[56] Hierzu gehört zum einen die Größe von Browsern resultierend aus dem hohen Implementierungsaufwand, zum anderen aber auch die Instabilität von Laufzeitumgebungen sowie große Anforderungen an Speicher- und Rechenleistung.

Ein nicht zu unterschätzendes Problem ist die starke Konkurrenz. Insbesondere Java3D sowie Adobe Flash konnten sich inzwischen gut etablieren. Java3D hat als API von Java diverse Vorteile gegenüber VRML. Dazu gehört auch, dass VRML nur eine Beschreibungssprache und keine höhere, geschweige denn objektorientierte Programmiersprache ist. Allerdings hat es die Möglichkeit, Java bzw. JavaScript über den Script-Knoten zu integrieren.

Große Popularität und Verbreitung im Web hat auch Flash erreicht. Die Anwendungsmöglichkeiten sind breit gefächert von Animationen über Filmen bis zu ganzen Internet-Seiten.

7.9 Fazit

„Auch wenn mit der ISO-Normierung zunächst die Dynamik der Entwicklung etwas beruhigt wurde, ist VRML mit seinen Möglichkeiten noch lange nicht am

[55] siehe ebda., S. 168 f.
[56] siehe K. Zeppenfeld, a.a.O., S. 393.

> Ende. Im Gegenteil, man kann erkennen, daß sich hinter den Kulissen allerhand tut. [...] Was und wieviel genau da auf uns zukommt, kann heute noch keiner vorhersehen — aber es ist klar, daß es nicht wenig sein wird. VRML hat sich einen Platz in der Computerwelt erobert, den es so schnell nicht wieder räumen wird.“[57]

Diese Zukunftsperspektiven wurden 1998 vorausgesagt — ein Jahr nach Erklärung von VRML zum ISO-Standard und ein Jahr bevor zum letzten Mal über eine neue Spezifikation von VRML diskutiert wurde, bevor die Weiterentwicklungen schließlich beim Nachfolger X3D endeten. Heutzutage ist nur noch Wenigen VRML ein Begriff. Die weite Verbreitung im Web verschwand; Homepages mit Welten oder Informationen wurden zum Teil vollständig ersetzt.

VRML konnte sich in relativ kurzer Zeit als Standard für 3D Webtechnologien etablieren und wurde sogar als ISO-Standard erklärt. Die Beschreibungssprache erzeugte genau das, was die Menschen sich damals wünschten — die dreidimensionale Darstellung im Web. Viele Firmen, selbst große wie Microsoft oder Apple, und auch Privatpersonen beteiligten sich an der Entwicklung. Zudem findet VRML in zahlreichen Gebieten Anwendung, insbesondere auch für Kunden, die beispielsweise ein Auto vorher im Web betrachten konnten, bevor sie sich auf den Weg zum Autohändler machen. Auch Städteplaner konnten mit Hilfe von VRML ein dreidimensionales Modell erstellen. Die Probleme, die VRML aufwies, konnten größtenteils gelöst werden oder es gab zumindest gute Aussichten auf baldige Lösung wie beispielsweise bei der Komplexität, wo man sich bereits Gedanken um ein binäres Dateiformat machte.

Der wohl ausschlaggebendste Grund, warum VRML letztendlich sich nicht weiter auf dem Markt behaupten konnte, ist, dass die anfängliche Dynamik in der Entwicklung durch die Erklärung als ISO-Standard verschwand, sei es aufgrund der zahlreichen Formalitäten, die mit der Änderung der Spezifikation verbunden waren oder sei es, das bloße Ausruhen auf Erfolgen, auf jeden Fall wurde dadurch der Konkurrenz ihre Möglichkeit gegeben. Der Mangel an Bekanntheit kann nicht an der Tatsache liegen, dass für VRML ein Browser oder PlugIn notwendig ist, denn das gleiche gilt für Flash. Darüber hinaus ist die Software zur Erstellung von Flash-Dateien kostenpflichtig, im Gegensatz zu VRML, das im Prinzip nur den Editor benötigt. Java3D zieht seine Vorteile aus der Verfügung über höhere Programmierlogik. Jedoch kann Java in VRML integriert werden. Trotzdem war die Konkurrenz stärker. Das kann unter anderem daran liegen, dass sowohl Java3D als auch Flash von großen Firmen stammen, die über bessere Mittel verfügen, was Marketing oder auch Entwicklung angeht. Ebenso sind Flash-Dateien eine gute Alternative zum Video, das zu viele Ressourcen benötigt.

Eindeutig ist jedenfalls, dass trotz der rosigen Zukunftsperspektiven die Ära VRML zu Ende ist, ganz im Gegensatz zur virtuellen Realität. Nichtsdestotrotz gibt es ein „Second Life“.

[57] O. Schlüter, a.a.O., S. 181.

Kapitel 8

Pathfinding und A*-Algorithmus

8.1 Einleitung

Zu den wohl anspruchsvollsten Anwendungen von Computergrafik zählen Computerspiele. Innerhalb dieses Gebietes gibt es viele Probleme, die ein Programmierer lösen muss, besonders wenn er ein anspruchsvolles Spiel erstellen will, in dem auch Computergegner auftauchen. Eines dieser Probleme ist das *Pathfinding*.

8.1.1 Was ist Pathfinding?

Grundsätzlich beschreibt der Begriff *Pathfinding* (dt. „Pfadfindung“) eine (computerbasierte) Methode, die das Finden eines Weges zwischen zwei Punkten auf einem gegebenen Suchraum ermöglicht. Typische Anwendungsgebiete neben der Netzwerkflussanalyse, Routenplanung und Robotik sind Computerspiele. Diese zeichnen sich vor allem durch intelligente, computergesteuerte Spielfiguren (*Bots*) aus, die die virtuelle Umgebung selbständig erforschen, z.B. auf der Suche nach dem Spieler (wie in einem Ego-Shooter) oder Ressourcen (in Strategiespielen).

Moderne Computerspiele werden zwar oft aufgrund ihrer audiovisuellen Elemente bewertet; wenn die computergesteuerten Gegner jedoch ständig gegen Hindernisse laufen oder einen „ungünstigen“ Weg einschlagen, verliert der Spieler schnell sein Interesse. Beschleunigt wird diese Desillusionierung, wenn eine Vielzahl von Bots berechnet werden muss und die Rechenleistung nicht mehr ausreicht. Die Verwendung schlechter Methoden beim Programmieren kann hierbei erheblich dazu beitragen. Pathfinding kann als ein wichtiger Teil der *künstlichen Intelligenz*[1] (KI) verstanden werden und ist entscheidend für die Verkaufszahlen eines Spiels.

[1] Wikipediazitat:„*Artificial Intelligence*; Teilgebiet der Informatik...mit Ziel der Entwicklung von Maschinen mit intelligentem Verhalten“

(a) Abbildung 1: Pacman

(b) Abbilddung 2: Age of Emprie III

8.1.2 Anforderungen an gutes Pathfinding

Im Spielebereich wird seit über 20 Jahren Pathfinding angewendet, um computergesteuerte Figuren intelligent agieren zu lassen. Eines der ersten und berühmtesten Spiele, die dieses Prinzip verwenden ist *Pacman*. [2] Trotz der grafischen Simplizität, oder gerade deswegen, verdeutlicht das Spiel aus den 80ern, welchen Problemen man sich bei dessen Entwicklung stellen muss: die Verfolgerfigur muss möglichst kurze Wege wählen, damit der Spieler Mühe hat, vor ihr zu fliehen. Dabei darf sie keine Wände durchlaufen oder ständig den gleichen Weg gehen.

In neueren Echtzeit-Strategiespielen, wie *Age of Empire 3*[3] , gibt der Spieler seinen militärischen Einheiten einen Zielpunkt vor. Die *Spiel-Engine*[4] berechnet dann den Weg, den die Figuren einschlagen. Doch auch hier müssen Hindernisse vermieden und umgangen werden. Dabei spielt z.B. das Gelände, in dem sich die Spielfigur bewegt, eine Rolle. Je nachdem, ob die Einheiten einen Wald, Sumpf oder eine offene Landschaft durchqueren, verändert sich ihre Geschwindigkeit, weil die damit verbundenen Wegekosten steigen. Deshalb ist es für den Programmierer zwingend notwendig eine geeignete Pfadfindungsstrategie auszuwählen, ohne dabei die Performance außer Acht zu lassen.

[2] 1979 von *Namico* veröffentlicht

[3] 2005 von *Ensemble Studios* entwickelt und von *Microsoft* veröffentlicht

[4] Wikipediazitat: „oder *Game-Engine* bildet das Grundgerüst der meisten Computerspiele. Sie besteht aus einer Programmbibliothek, die Entwicklern von Computerspielen häufig benutzte Werkzeuge zur Verfügung stellt.“

8.2 Strategien für Pathfinding

In gegenwärtigen Spielen wird Pathfinding meist als zweiphasiger Prozess realisiert. Neben der Berechnung der Wege zur Laufzeit ist es nötig im Vorfeld die virtuelle Landschaft im *Preprozessing*[5] zu analysieren. Dabei werden unveränderliche Teile des Terrains wie z.B. Gebirge, Flüsse oder Seen untersucht und die draus resultierenden in Frage kommenden Wege für eigentliche Wegberechnung zwischengespeichert. Bei großen Landschaftskarten kann das Preprozessing mehrere Minuten dauern, weshalb die Lösung meist vom Hersteller fertig mitgeliefert.

8.2.1 Schritte für Suchverfahren

Damit ein Pathfinding-Algorithmus auf einer Karte angewendet werden kann, muss diese in eine logische, für den Algorithmus verständliche Repräsentation überführt werden. Da die Pathfinding-Algorithmen ihren Ursprung in der *Graphentheorie*[6] haben, besteht die Repräsentation üblicherweise aus einem ungerichteten Graphen mit Knoten, die die Wegpunkte markieren. Aus diesem Grund ist es notwendig, die Karte zu unterteilen und die daraus resultierenden Bereiche (Knoten im Graphen) miteinander zu verbinden. Eine Möglichkeit für diese Partitionierung ist die Zerlegung in ein gleichmäßiges Raster, z.B. in Quadrate. Jedes dieser Quadrate bezeichnet dann einen Knoten im Graphen. Wenn diese nebeneinander liegen und man von einem das andere erreichen kann, werden die dazugehörigen Knoten mit einer Kante verbunden. Ihren häufigsten Einsatz findet diese Methode in Strategiespielen.

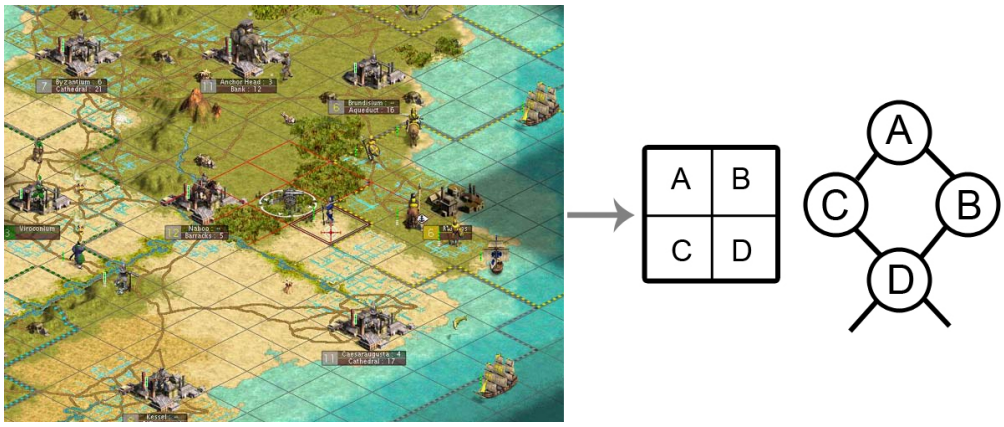


Abbildung 3: Unterteilung der Landschaft in ein Raster

Eine weitere Möglichkeit, die virtuelle Umgebung für die Wegberechnung zu unterteilen, besteht darin, Wegpunkte zu setzen. Dabei werden, in Abhängigkeit von der Gestaltung des Raumes,

[5] Vorberechnung statischer Geländedetails

[6] Wikipediazitat:„Teilgebiet der Mathematik, das die Eigenschaften von Graphen und ihre Beziehungen zueinander untersucht.“

Punkte erzeugt, die die Figur begehen kann. Diese können durch Berechnung von Distanzen zu Hindernissen gesetzt werden. Dabei kann eine Vielzahl von unbegehbaren Stellen ausgespart werden, wodurch der logische Graph stark reduziert und die Wegsuche deutlich beschleunigt wird. Diese Methode wird oft in Ego-Shootern benutzt. Ein Nachteil jedoch ist, dass die Wege für den Spieler vorhersehbar werden, da die Figur innerhalb eines Raumes die gleichen Positionen einnehmen kann.

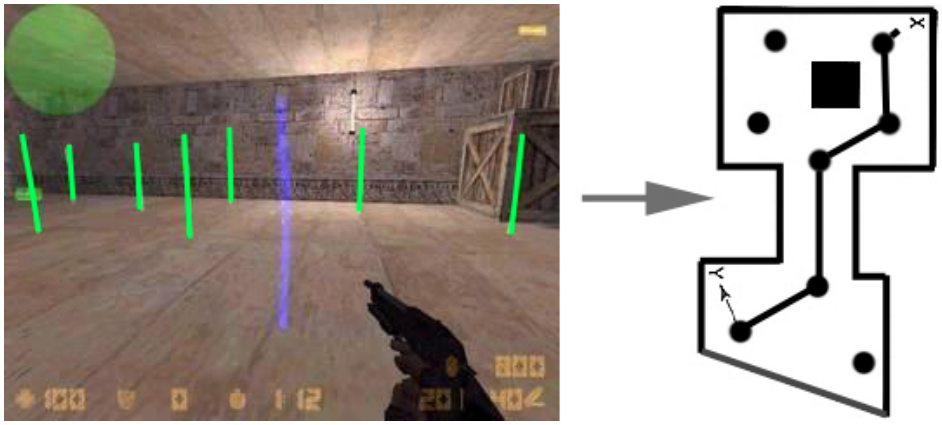


Abbildung 4: Wegepunkte-Graphen

8.2.2 Uninformierte vs informierte Suchverfahren

Wurde die Karte in Teilbereiche unterteilt und diese in die logische Repräsentation, d.h. einen Graphen, überführt, kann nun ein Pathfinding-Algorithmus angewendet werden. Um kürzeste Wege innerhalb eines Graphen zu berechnen, hat man mehrere Möglichkeiten. Wir unterscheiden zwei grundlegenede Methoden: *uninformierte Suche* und *informierte Suche*.

Die Algorithmen der uninformierten Suche können eine Lösung nur durch systematisches und erschöpfendes Ablaufen des Graphen erreichen, da sie keine Information darüber haben, wo sich das Ziel befindet. Dadurch sind sie meist sehr ineffizient, eignen sich aber sehr gut, um die Theorie der Suchalgorithmen zu verstehen, da sie schnell und einfach zu implementieren sind. In modernen Computerspielen haben diese Algorithmen jedoch keine Relevanz, weil bei ihnen die Möglichkeit von Wegekostenberücksichtigung fehlt.

Informierte Suchverfahren nutzen hingegen problem-spezifisches Wissen, um die Reihenfolge der Knotenexpansion zu steuern und die Anzahl der Knoten zu beschränken. Sie haben einen entscheidenden Vorteil gegenüber den uninformierten Algorithmen, da sie den kürzesten Weg schneller finden.

8.3 Algorithmen für Pathfinding

8.3.1 Uninformierte Algorithmen

Zu den uninformierten Algorithmen ohne Berücksichtigung von Wegekosten zählen die *Breitensuche* und die *Tiefensuche*. Als eine Erweiterung der Breitensuche kann der Algorithmus von Dijkstra angesehen werden, der es erlaubt terrain-spezifische Kosten zu implementieren, die eine Spielfigur in Kauf nehmen muss, wenn sie z.B. durch einen Sumpf läuft.

8.3.1.1 Breitensuche

Die Breitensuche (engl. *breath-first-search*) durchsucht den Suchgraphen systematisch. Sie beginnt mit dem Startknoten und endet, wenn sie den Zielknoten gefunden hat. Dabei werden weiter oben liegende Knoten bevorzugt, d.h. sie bearbeitet zunächst alle nachfolgenden Knoten innerhalb einer Ebene. Die Datenstruktur, in der sich die Knoten befinden, kann als eine Warteschlange (engl. *queue*, FIFO-Speicher) angesehen werden.

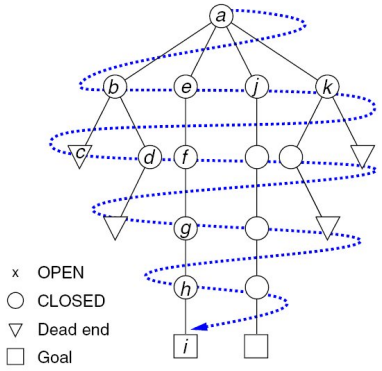


Abbildung 5: Suchdurchlauf der Breitensuche

Der Algorithmus der dieser Suche zugrunde liegt ist sehr einfach und kann wie folgt in einem Flussdiagramm dargestellt werden:

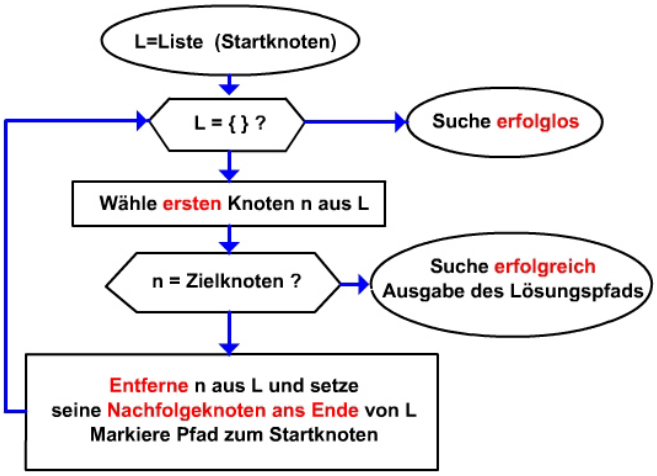


Abbildung 6: Flussdiagramm der Breitensuche

Diese Suche findet im besten Fall den Zielknoten am schnellsten, wenn dieser der erste Knoten der unter dem Startknoten liegenden Ebene ist. Im schlechtesten Fall ist der Zielknoten der letzte der untersten Ebene. Für diesen Fall kann man folgende Laufzeit berechnen, wobei b die Breite und d die Tiefe ist:

$$\text{Kosten für vorangegangene Ebene: } \sum_{k=0}^{d-1} b^k = \frac{(b^d-1)}{(b-1)}$$

Kosten der vorangegangenen Ebene + Kosten der aktuellen Ebene:

$$\frac{(b^d-1)}{(b-1)} + b^d = \frac{(b^d-1)}{(b-1)} + \frac{b^d*(b-1)}{(b-1)} = \frac{(b^{(d+1)}-1)}{(b-1)} = O(b^d)$$

8.3.1.2 Tiefensuche

Die Tiefensuche (engl. *depth-first-search*) erkundet den Graphen ebenfalls systematisch, jedoch werden tiefer liegende Knoten bevorzugt, d.h. als Folgeknoten wird der untere Knoten gewählt und nicht der in gleicher Ebene liegende. Dies ermöglicht einen rekursiven Aufruf, in dem die Knoten in der Datenstruktur eines Kellerspeichers (engl. *stack*, FILO- Speicher) abgelegt werden. Diese Methode hat gegenüber der Breitensuche den Vorteil eines geringeren Speicherbedarfs. Als Nachteil ist hierbei zu nennen, dass auch sinnlose Wege berücksichtigt werden und bei ungerichteten Graphen mit Wiederholungen keine Terminierung möglich ist und der Algorithmus somit nie endet. Das Flussdiagramm ist mit dem der Breitensuche fast identisch, nur werden die Nachfolgeknoten an den Anfang und nicht ans Ende der Liste gesetzt.

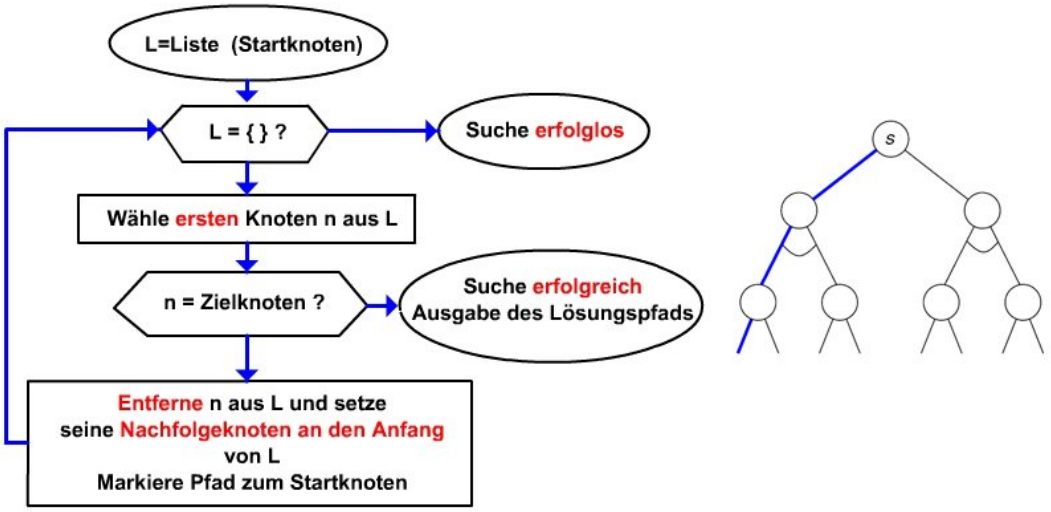


Abbildung 7: Flussdiagramm der Tiefensuche

Dieser Suchalgorithmus findet im besten Fall sofort den Zielknoten, wenn sich dieser als erster im linken Teil des Baumes befindet. Folgende Laufzeit kann man dafür errechnen:

$$d(b-1)+1 = O(b*d) \text{ Linear!}$$

Der schlechteste Fall tritt auf, wenn der Zielknoten im untersten, rechten Teil des Baumes liegt. Die Laufzeit hierfür ist wie bei der Breitensuche:

$$\sum_{k=0}^{d} b^k = \frac{(b^{(d+1)}-1)}{(b-1)} = O(b^d)$$

8.3.1.3 Algorithmus von Dijkstra

Im Gegensatz zu der Breiten- und Tiefensuche ist es mit Hilfe des *Dijkstra-Algorithmus*[7] möglich, einen Graphen mit Kanten unterschiedlicher Gewichtung zu durchsuchen. Dadurch können Wegekosten für die Begehung von unterschiedlichem Terrain implementiert werden. Bewegt sich die Figur durch Wald oder sumpfiges Gelände, entstehen dadurch größere Wegekosten, die eingespart werden können, wenn die Figur stattdessen einen möglicherweise längeren Weg einschlägt, der jedoch schneller zum Ziel führt. Des Weiteren ist es möglich Wege auszuwählen,

[7]Edsger Wybe Dijkstra (geb.: 11. Mai 1930 in Rotterdam; gest.: 6. August 2002 in Nuenen, Niederlande); einflussreicher niederländischer Informatiker

die aufgrund fehlender gegnerischer Einheiten strategisch sinnvoller erscheinen. Was die Kosten im Endeffekt verursacht, ist dem Programmierer und der Notwendigkeit des Spiel überlassen.

Die Umsetzung basiert hierbei auf *Prioritätswarteschlangen*, wobei die Knoten nach dem bisher insgesamt aufgewendeten Preis in der Schlange sortiert werden. Diesen Algorithmus bezeichnet man als gierig (engl. *greedy*), da der beste Knoten mit den geringsten Kosten im aktuellen Zustand zur weiteren Betrachtung herangezogen wird. Grundlegend kann man sagen, dass der Dijkstra-Algorithmus dem der Breitensuche sehr ähnlich ist und immer den optimalen Weg schnell findet. Gleichsam besteht ein ähnlich hoher Speicherbedarf und die Implementierung ist etwas schwieriger. Am Flussdiagramm lässt sich dies gut erkennen.

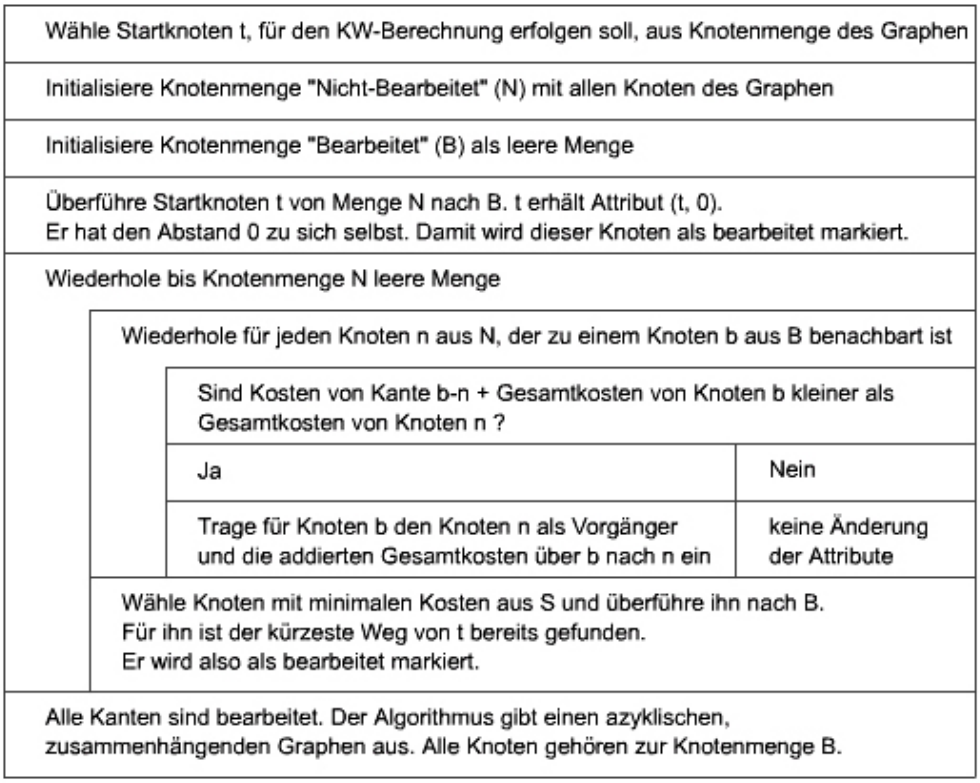


Abbildung 8: Flussdiagramm des Dijkstra-Algorithmus [8]

8.3.2 Informierte Algorithmen

In vielen Fällen lässt sich die Knotenexpansion auf der Suche nach dem Ziel beschleunigen, indem man Wissen über die mögliche Lage des Zielknotens einsetzt. Dabei werden nicht nur die Kosten berücksichtigt, die bis zu dem aktuellen Knoten entstanden sind, sondern eine Schätzung über die Restdistanz während der Laufzeit angefertigt, die als *Heuristik* bezeichnet wird. Eine

[8] Quelle: Uni Stuttgart

Heuristik ist eine Arbeitshypothese als Hilfsmittel der Forschung, mit der die Suche in Richtung des Zielknotens gelenkt wird. Die wohl bekannteste und meist genutzte Methode ist der im Jahre 1960 von Peter Hart, Nils Nillson und Bertram Raphael vorgestellte A*-Algorithmus[9].

8.3.2.1 Der A*-Algorithmus

Der A*-Algorithmus ähnelt dem Algorithmus von Dijkstra, jedoch wird zur näheren Betrachtung zusätzlich zu den bisherigen geringsten Kosten der Knoten herangezogen, der durch die Heuristik-Funktion $h(n)$ am wahrscheinlichsten den geringsten Restaufwand zum Ziel hat. Es müssen weniger Knoten kontrolliert werden, da die Restkosten steigen, wenn man sich vom Ziel entfernt. Die Kosten-Funktion setzt sich somit aus den Kosten für die zurückgelegte Distanz $g(n)$ und dem geschätzten Restaufwand für die Distanz vom aktuellen Knoten zum Ziel $h(n)$ zusammen.

$$\text{Gesamtkosten: } f(n) = g(n) + h(n)$$

Diese Methode funktioniert besonders gut, weil die Abschätzung entweder kleiner oder gleich der eigentlichen Entfernung ist, d.h. die Heuristik ist dann zulässig, wenn sie die tatsächlichen Kosten nicht überschätzt. Meist handelt es sich dabei um die Luftlinie von der aktuellen Position zum Zielpunkt. Zur Implementierung des Algorithmus werden zwei Listen benötigt. Die eine Liste (*closed-list*) enthält die bereits besuchten Knoten, die andere (*open-list*) enthält dementsprechend die Knoten, die noch nicht besucht worden sind. Bei jedem Schritt wird der Knoten herangezogen, der die geringsten Gesamtkosten hat. Zu Beginn ist es der Startknoten. Danach werden alle Nachbarknoten untersucht und der Startknoten selbst wird von der open-list in die closed-list gesetzt. Zur weiteren Betrachtung wird zuerst der Nachbarknoten mit geringsten Gesamtkosten herangezogen, falls dieser noch nicht untersucht wurde. Es ist notwendig jedes Mal wenn ein Knoten einen kürzeren Weg garantiert, die Kosten für seine Nachbarknoten neu zu berechnen. Sollte die open-list leer sein und das Ziel nicht gefunden werden, existiert kein Weg vom Start- zum Zielpunkt. Zusammenfassend lässt sich folgender Pseudocode für den Algorithmus angeben:

- setze Startknoten in open-list
- wiederhole
  - suche Knoten mit minimalen Gesamtkosten
  - verschiebe ihn in closed-list
  - für jeden Nachbarknoten (begehbar und nicht in closed-list)
    - wenn nicht in open-list dann füge hinzu

[9] gesprochen: A Stern

    - aktueller Knoten als Vorgänger eintragen mit f,g und h Kosten
    - wenn in open, ist neuer Pfad besser?
    - wenn ja, dann Vorgängerknoten auf aktuellen Knoten setzten (neu berechnen)

- beende, wenn Ziel in closed-list oder kein Ziel gefunden und open Liste leer

Besonders gut kann man anhand des folgenden Beispiels sehen, wie dieser Algorithmus arbeitet. Die Karte besteht aus Quadraten, von denen einer das Startquadrat (grün) und ein anderer das Zielquadrat (rot) ist. Die blauen Quadrate repräsentieren Hindernisse, die umgangen werden müssen.

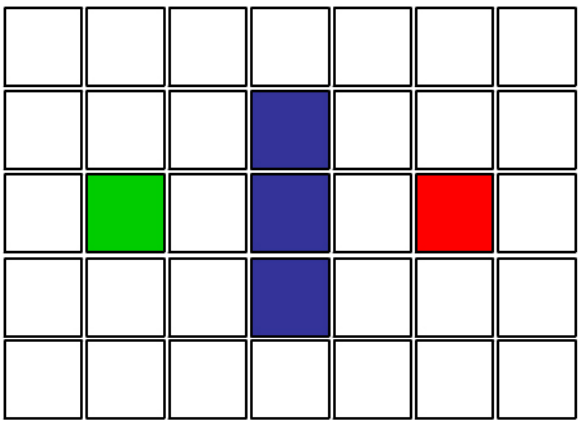


Abbildung 9: A*-Anwendung

Zuerst werden die Nachbarquadrate des Startquadrats untersucht und die Kosten berechnet. Bewegt sich eine Spielfigur auf diesem Feld in horizontaler oder vertikaler Richtung, entspricht das einem Kostenwert von 10 und in diagonaler Richtung einem Wert von 14 (links unten im Quadrat). Die heuristische Schätzfunktion, die hier verwendet wird, ist der Abstand von einem Quadrat zum Ziel, wobei nur vertikale oder horizontale Bewegung erlaubt ist. Dabei werden Hindernisse außer Acht gelassen. Diese Schätzung wird als *Mannhatan-Heuristik* bezeichnet und ihr Wert befindet sich rechts unten im jeweiligen Quadrat. Die Gesamtkosten ist die Summe der beiden Werte und wird links oben dargestellt.

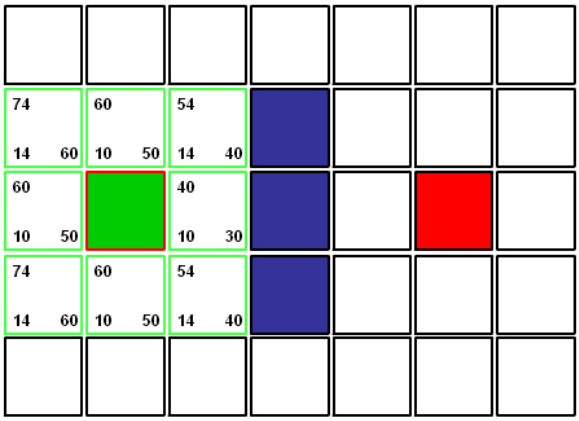


Abbildung 10: A*-Anwendung

Wendet man systematisch den oben beschriebenen Algorithmus an, wird schnell der Weg mit niedrigsten Kosten gefunden.

Der A*-Algorithmus enthält die klassischen Algorithmen von Dijkstra, falls $k \geq 0$ und $h \equiv 0$ und für $k \equiv -1$ und $h \equiv 0$ die Tiefensuche.

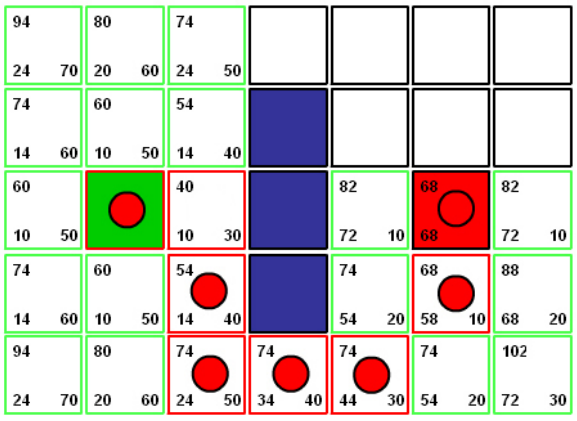


Abbildung 11: A*-Anwendung

Ein großer Nachteil vom A*-Algorithmus ist, dass er relativ viel Rechenleistung der CPU verbraucht, besonders wenn Wege für viele Charaktere berechnet werden müssen.

Eine Lösung für dieses Problem ist der *A* mit iterierter Tiefensuche (IDA*)*, eine Tiefensuche die $f(n) = g(n) + h(n)$ statt der Tiefe $d(n)$ als Schranke verwendet, die in ε-Schritten erhöht wird. Darüberhinaus existieren weitere Implementierungen, die z.B. den Speicherbedarf reduzieren oder schneller auf dynamische Veränderungen reagieren.

8.4 Literaturangabe

- www.policyalmanac.org
- http://theory.stanford.edu
- Uni Stuttgart
- http://www.gutschke.de
- http://www.cs.utexas.edu/users/EWD
- http://raskob.info
- http://electures.informatik.uni-freiburg.de
- http://en.wikipedia.org/wiki/Pathfinding
- Introduction to algorithms, von Thomas H.Corman ISBN 0-07-013151-1 (McGraw-Hill)
- Algorithmen und Datenstrukturen, von T.Ottmann und P.Widmayer ISBN 3-8274-1029-0
- Game programming Gems 3, von Dante Treglia ISBN 1-58450-233-9

Kapitel 9

Delaunay-Triangulierungen

9.1 Was ist eine Triangulierung?

Aus früheren Vorträgen ist uns bekannt, dass Objekte in der 3D-Computergrafik in der Regel durch Dreiecke dargestellt werden. In diesem Artikel betrachten wir folgendes damit verbundene Problem:

Angenommen, wir verfügen über eine Menge von Eckpunkten (Punktewolke) eines dreidimensionalen Objekts. Nun möchten wir diese Punkte automatisch so zu Dreiecken verbinden, dass das Objekt mit einer Haut überzogen wird, die wir anschließend einfärben oder texturieren können, um es als massiven Körper darzustellen. Ein solches Netz von zusammenhängenden Dreiecken nennt man eine **Triangulierung**.

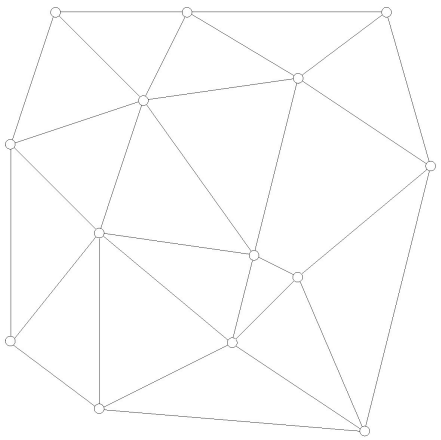


Abbildung 9.1: Eine typische, einfache Triangulierung

9.1.1 Anwendungsbeispiele

Eine typischer Einsatzbereich für Triangulierungsalgorithmen ist die Darstellung von Gelände in 3D-Anwendungen. Ein Beispiel ist der Open-Source-Flugsimulator **FlightGear**: Dessen Szenerielandschaften werden automatisch durch Triangulierung von geografischen Koordinatenpunkten erzeugt, deren Geländehöhe in Form der SRTM (Shuttle Radar Topography Mission)-Daten für die gesamte Erde frei verfügbar sind. Ebenfalls automatisch in die Triangulierung eingebunden werden Vektorkarten über die Landflächennutzung (Bebauung, Wald, Landwirtschaft, Gewässer etc.) sowie die größten Straßen und Eisenbahnstrecken. Das Ergebnis sieht so aus:

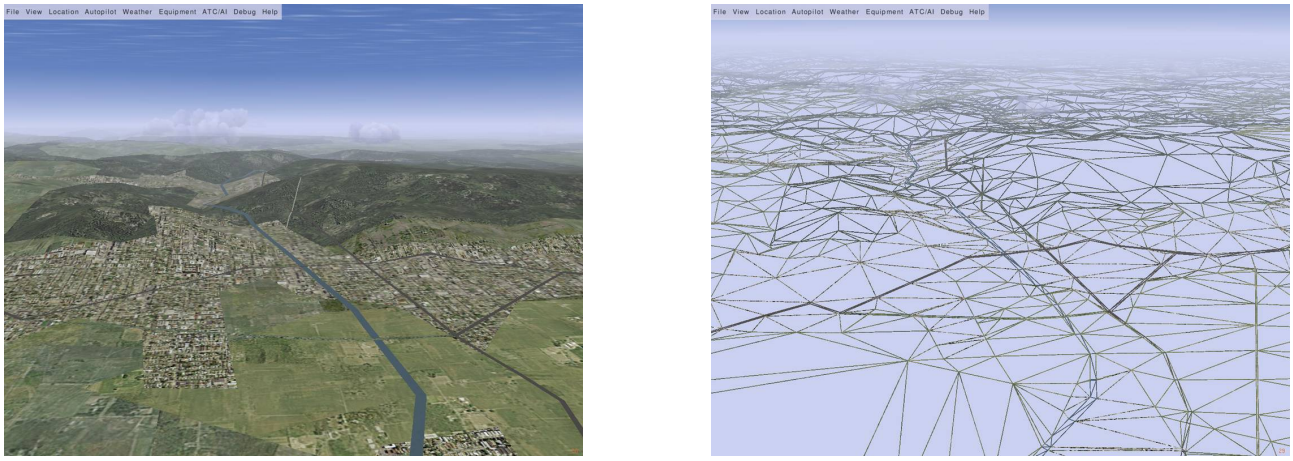


Abbildung 9.2: Dreiecksbasierte Darstellung der Landschaft in FlightGear - links texturiert, rechts als Wireframe-Ansicht zur Veranschaulichung des Aufbaus

Nach dem gleichen Prinzip können auch beliebige Mess- oder Rechenergebnisse als dreidimensionale Diagramme visualisiert werden. So könnte man etwa mittels einer Triangulierung die Temperaturverteilung auf einer Oberfläche als „hügelige Landschaft“ über dieser Oberfläche darstellen, wobei die Höhe eines Punktes der Temperatur an der jeweiligen Stelle entspricht.

Auch zur dreidimensionalen Erfassung realer Objekte („3D-Scannen“) werden Triangulierungsalgorithmen benutzt. Ein 3D-Scanner erfasst die Form eines Körpers und speichert die Raumkoordinaten einer gewissen Menge von Punkten auf seiner Oberfläche. Je mehr Punkte vermessen werden, desto höher wird die Qualität des 3D-Modells. Durch Triangulierung wird diese Punktewolke anschließend mit einer deckenden „Haut“ aus Dreiecken überzogen, die letztlich noch eingefärbt oder texturiert werden kann, um den Körper realistisch darzustellen.

9.1.2 Was ist eine Delaunay-Triangulierung?

Offensichtlich gibt es für eine gegebene Punktemenge im Allgemeinen zahlreiche Möglichkeiten, sie zu triangulieren. Gibt es eine Triangulierung mit bestimmten besonderen Eigenschaften, durch die sie für Zwecke wie die oben besprochenen Beispiele besonders gut geeignet ist?

Zwischenbemerkung:

Viele der im Folgenden betrachteten Konzepte gelten unmittelbar oder mit Abwandlungen auch für Räume mit **mehr als zwei Dimensionen**. Wir gehen in diesem Artikel allerdings immer vom zweidimensionalen euklidischen Raum R^2 aus und behandeln lediglich die auf diesen Raum eingeschränkten Fälle.

Dies ist **kein Widerspruch** zu dem Ziel, die Triangulierung für „2,5"-dimensionale Strukturen wie etwa das oben genannte Beispiel Landschaftsdarstellung einzusetzen, da die Z-Koordinate (Höhe eines Punktes) für die Triangulierung unwesentlich ist.

Um die oben gestellte Frage zu beantworten, stellen wir folgende Überlegung an:

- Das zu triangulierende Punktenetz ist eine **Annäherung**, ein vereinfachtes Modell der nachzubildenden tatsächlichen Form.
- Exakt platziert sind nur die duch die Punktemenge gegebenen **Eckpunkte**. Alle Oberflächenpunkte dazwischen, die auf den erzeugten Dreiecksflächen liegen, werden **interpoliert**.
- Je größer der Abstand eines Punktes von einem Eckpunkt „seines" Dreiecks ist, desto höher ist der **potentielle Fehlerfaktor**, d.h. die Abweichung vom tatsächlichen (Höhen)-wert. Unser 3D-Modell soll die tatsächliche Form aber **möglichst genau** wiedergeben.

Wir möchten also eine Triangulierung, bei der **der durchschnittliche Abstand aller Punkte einer Dreiecksfläche vom Nächsten ihrer Eckpunkte möglichst gering ist**, denn dadurch werden die interpolationsbedingten Abweichungen minimiert.

Erfüllt wird diese Anforderung von der sogenannten **Delaunay-Triangulierung**. Sie ist benannt nach dem russischen Mathematiker *Boris Nikolaevich Delone* (1890 - 1980, französische Form des Nachnamens: Delaunay). Die Delaunay-Triangulierung ist folgendermaßen definiert:

Definition: Eine Verbindung zwischen zwei Punkten P,Q einer Punktemenge S heißt **Delaunay-Kante** genau dann, wenn ein Kreis K existiert, der P und Q einschließt, aber kein anderer Punkt aus S innerhalb von K liegt.

Definition: Eine Triangulierung heißt **Delaunay-Triangulierung** genau dann, wenn jede ihrer Kanten eine Delaunay-Kante ist bzw. wenn für jedes Dreieck der Triangulierung gilt, dass innerhalb seines Umkreises kein Eckpunkt eines anderen Dreiecks liegt. Mann nennt dies die **Umkreisbedingung**.

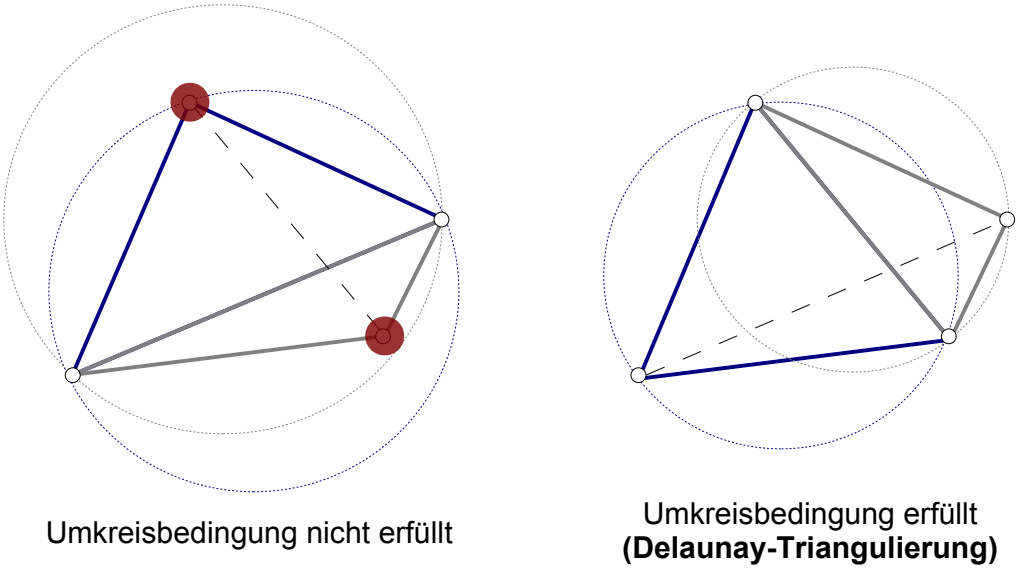


Abbildung 9.3: Unterschiedliche Triangulierungen von vier Punkten

9.1.3 Das Voronoi-Diagramm

Das Voronoi-Diagramm ist eine geometrische Struktur mit engem Bezug zur Delaunay-Triangulierung. Benannt ist es nach dem russischen Mathematiker Georgi Woronoi (1868-1908). Es ist auch bekannt als Thiessen-Diagramm oder Dirichlet-Tesselation.

Definition: Sei P eine Punktemenge im R^2. Die Menge aller Punkte des R^2, die näher an einem Punkt $p_i \in P$ liegen als an jedem anderen Punkt aus P, heißt die **Voronoi-Zelle** zu p_i:

$V_i = \{x \in R^2 | \ \forall \ j \neq i : d(x, p_i) \leq d(x, p_j)\}$

Definition: Die disjunkte Vereinigung aller Voronoi-Zellen zu einer Punktemenge P heißt das **Voronoi-Diagramm** zu P.

Man nennt das Voronoi-Diagramm zu einer Punktemenge P den **dualen Graph** zur Delaunay-Triangulierung von P, denn:

Die **Ecken der Voronoizellen** sind die **Umkreismittelpunkte der Dreiecke** der Delaunay-Triangulierung.

Daraus ergibt sich, dass man über ein gegebenes Voronoi-Diagramm zu einer Punktemenge leicht eine Delaunay-Triangulierung dieser Punktemenge erhalten kann (siehe auch Abschnitt 2.5).

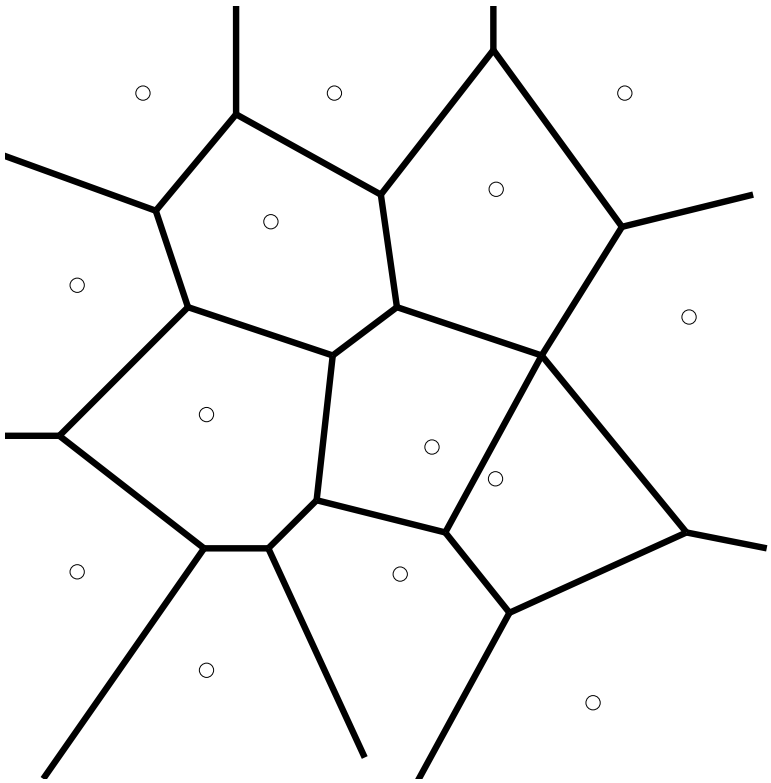


Abbildung 9.4: Voronoi-Diagramm

9.1.4 Anwendungsbeispiele für Voronoi-Diagramme

Das Konzept des Voronoi-Diagramms ist vielfältig anwendbar. In der Computergrafik lässt es sich beispielsweise zur Erzeugung von prozeduralen Texturen wie Schuppen- oder Steinplattenmustern oder „Mosaik“-Bildfiltereffekten nutzen.

Geometrische Anwendungen abseits der Computergrafik wären beispielsweise:

- **Berechnung des Pfades eines Roboters durch ein Feld von Gefahrenpunkten:**

  Der Roboter hat dabei die Vorgabe, von diesen maximalen Abstand zu halten. Dazu erzeugt man das Voronoi-Diagramm zu der Menge der Gefahrenpunkte und gibt dem Roboter die Anweisung, sich immer entlang der Zellenkanten zu bewegen.

- **Wahl des Standortes für einen neuen Supermarkt/Überwachungskamera/Wetterstation/**

  Man berechnet das Voronoi-Diagramm für die Punktemenge der Standorte der bereits vorhandenen Objekte. Das neue Objekt wird nun vorzugsweise an einem Eckpunkt dreier Voronoi-Zellen platziert, dessen mittlerer Abstand zu den nächsten Standorten möglichst hoch ist. Auf diese Weise wird die Effizienz des neuen Objekts maximiert.

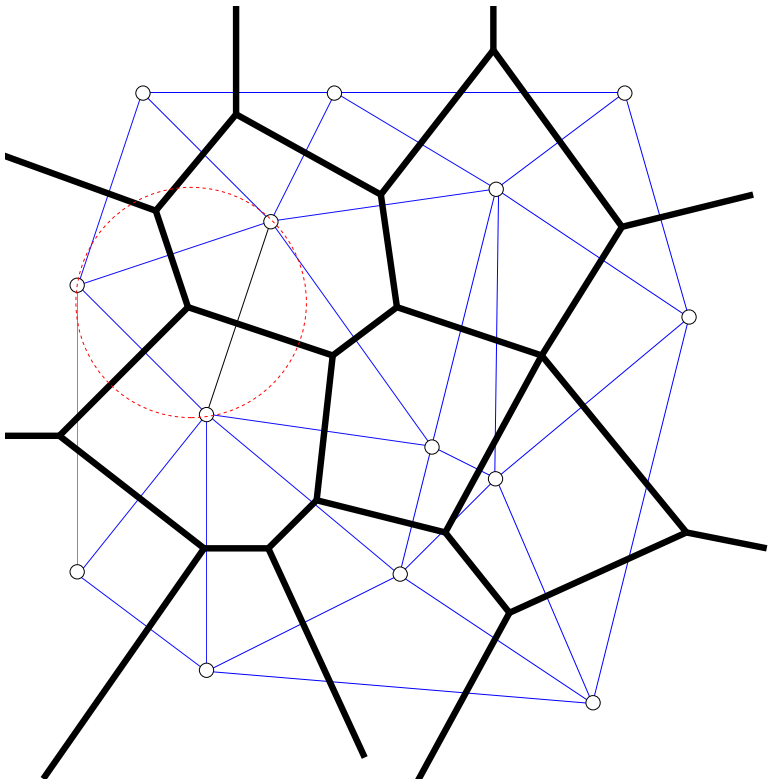


Abbildung 9.5: Dualität zwischen Voronoi-Diagramm und Delaunay-Triangulierung

9.2 Algorithmen zur Erzeugung von Delaunay-Triangulierunge

Zur Berechnung von Delaunay-Triangulierungen existieren zahlreiche Algorithmen. Die meisten sind sich allerdings sehr ähnlich, und fast alle basieren letztendlich auf der Überprüfung und Herstellung der lokalen Delaunay-Eigenschaft durch Kantentausch (Edge Flipping). In diesem Kapitel werden wir uns einen kurzen Überblick über einige der gebräuchlichsten Methoden verschaffen.

Generell unterscheidet man bei den Delaunay-Triangulierungsalgorithmen zwei Gruppen: **Statische** und **dynamische** Algorithmen. Die **statischen Algorithmen** sind dadurch charakterisiert, dass mit einer Triangulierung begonnen wird, die von Anfang an alle Punkte enthält, allerdings i.d.R. noch nicht global die Delaunay-Eigenschaft erfüllt. Die Aufgabe des Algorithmus besteht dann darin, die gegebene Triangulierung so zu verändern, dass sie global-delaunay wird.

Bei den **dynamischen Algorithmen** wird im Gegensatz dazu die gewünschte Delaunay-Triangulierung durch schrittweises Erweitern Punkt für Punkt aufgebaut. Die entstehende Triangulierung ist dabei von Anfang an global-delaunay.

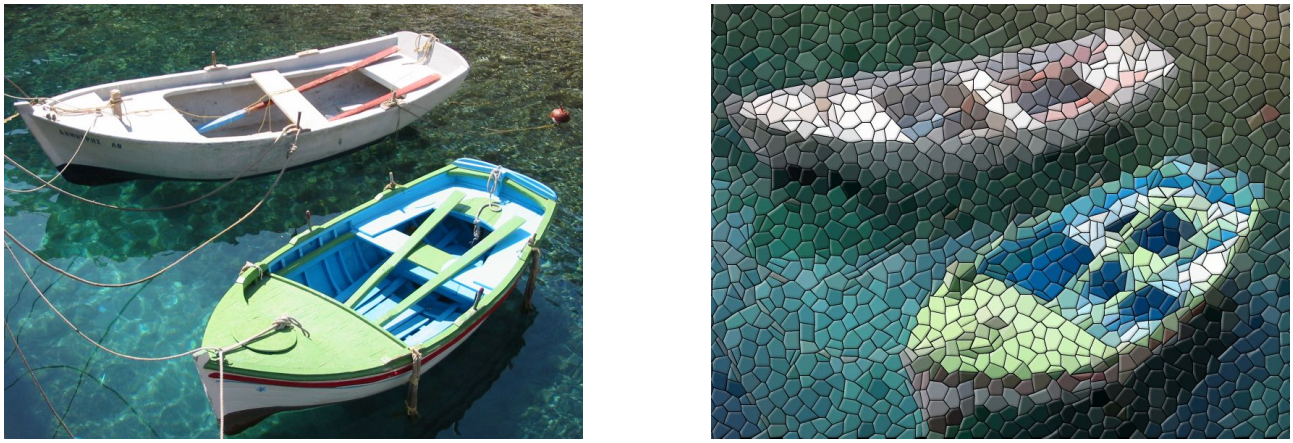


Abbildung 9.6: Mosaik-Effektfilter als Anwendungsbeispiel für Voronoi-Diagramme

9.2.1 Edge Flipping

Die Idee ist simpel: Wir beginnen mit einer bereits gegebenen, beliebigen Triangulierung und überprüfen für alle Paare von benachbarten Dreiecken die Umkreisbedingung. Falls sie nicht erfüllt ist, stellen wir sie her. Folglich zählt dieses Verfahren zu den dynamischen Algorithmen.

Der Vorteil dieser Methode ist ihre einfache Implementierung. Nachteilig ist das schlechte Laufzeitverhalten von $O(n^2)$ und der Umstand, dass sie ohne Erweiterungen nur im zweidimensionalen Raum verwendbar ist.

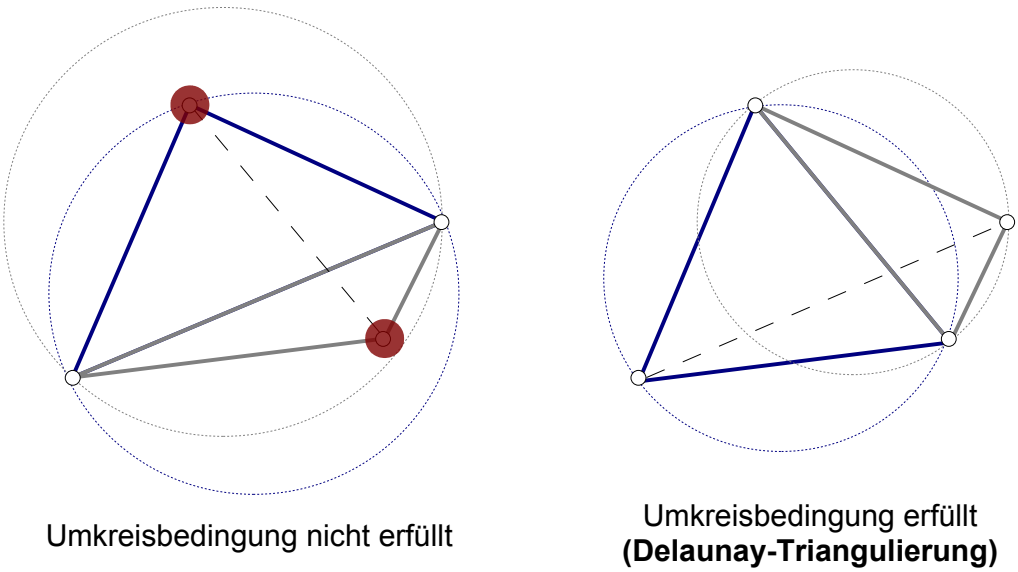


Abbildung 9.7: Herstellung der lokalen Delaunay-Eigenschaft durch Kantentausch

9.2.2 Incremental construction

Incremental Construction ist ein Beispiel für einen dynamischen Algorithmus. Er funktioniert folgendermaßen:

Zuerst erzeugen wir (gegebenenfalls durch Hinzufügen zusätzlicher Punkte) ein Dreieck, das alle zu triangulierenden unkte einschließt. Dieses eine Dreieck erfüllt trivialerweise die Umkreisbedingung.

Anschließend werden die zu triangulierenden Punkte Schritt für Schritt hinzugefügt. Dazu wird das jeweilige Dreieck, in dem sich ein neu hinzugefügter Punkt befindet, durch Einfügen von Kanten von seinen Eckpunkten um eingefügten Punkt in drei neue Unterdreiecke aufgespalten und gegebenenfalls durch Kantentausch die lokale Delaunay-Eigenschaft hergestellt. Dieser Vorgang wird so lange wiederholt, bis alle zu triangulierenden Punkte „eingebaut“ sind. Abschließend müssen gegenebenfalls noch die anfangs zum Aufspannen des allumfassenden Ausgangsdreiecks zusätzlich eingefügten Punkte wieder entfernt werden.

Vorteilhaft an dieser Methode ist erneut die einfache Implementierung, im Gegensatz zum naiven Edge Flipping lässt sie sich jedoch auf beliebig viele Dimensionen verallgemeinern.

Der Nachteil ist, wie beim Edge Flipping, das schlechte Laufzeitverhalten von $O(n^2)$

9.2.3 Sweepline

Bei Sweepline handelt es sich um eine Variation von Incremental Construction. Das Prinzip ist das gleiche, es wird lediglich eine Regel für die Einfügereihenfolge neuer Punkte festgelegt, während beim einfachen Incremental Construction diesbezüglich nichts vorgeschlagen wird. Die Einfügereihenfolge ergibt sich durch das Führen einer sogenannten „Sweepline“ über die zu triangulierende Punktemenge. Die Sweepline ist eine Gerade (für zweidimensionale Triangulierungen - für dreidimensionale eine Ebene, für vierdimensionale ein Quader usw.), welche z.B. von links nach rechts über die Punkteebene geführt wird. Es gilt dabei die Regel, dass ein neuer Punkt dann in die Triangulierung aufgenommen wird, sobald er von der Sweepline überfahren wird. Damit bilden alle Punkte links der Sweepline zu jedem Zeitpunkt eine globale Delaunay-Triangulierung, während rechts davon liegende Punkte noch nicht aufgenommen sind.

9.2.4 Divide and Conquer

Die grundlegende Idee eines „Divide and Conquer“ (Teile und Herrsche) -Algorithmus ist es, ein Problem zur Lösung in mehrere kleine Teilprobleme zu zerlegen, um einen Geschwindigkeitsvorteil aus dem Umstand zu erzielen, dass sich diese kleineren Teilprobleme einzeln für sich, dadurch im Endeffekt aber auch in ihrer Gesamtsumme, schneller lösen lassen als wenn man das gesamte Problem als Einheit behandelt. Dieses Prinzip funktioniert mit vielen typischen Aufgaben der Informatik, ein sehr bekanntes Beispiel ist etwa der Sortieralgorithmus QuickSort.

Divide and Conquer im Bezug auf die Delaunay-Triangulierung bedeutet, die Punktemenge in mehrere kleine Teilmengen zu zerlegen, diese zunächst separat zu triangulieren und schließlich die resultierenden Teiltriangulierungen rekursiv zusammenzufügen. Konkret geht man folgedermaßen vor:

Zu Beginn sortieren wir die zu triangulierenden Punkte entlang einer beliebigen Koordinatenachse. Anschließend teilen wir die Punktemenge entlang dieser Achse so lange rekursiv, bis nur noch Teilmengen mit vier oder weniger Punkten vorliegen. Diese kleinen Teilmengen triangulieren wir nun mit einem beliebigen Verfahren (z.B. Edge Flipping). Anschließend verbinden wir je zwei der auf diese Weise erzeugten Teiltriangulierungen durch Triangulierung ihres Zwischenraums. Auf diese Weise entstehen größere Triangulierungen, die erneut auf die gleiche Weise verbunden werden. Dies wiederholen wir so lange rekursiv, bis wir eine einzige Delaunay-Triangulierung erhalten haben, die alle Punkte enthält.

Da der Aufwand für das Zusammenfügen zweier Teiltriangulierungen im ungünstigsten Fall $O(n)$ beträgt, ergibt sich ergibt sich einschließlich des Vorsortierens der Punkte die für Divide-and-Conquer-Algorithmen typische Laufzeitkomplexität von $O(n*log(n))$. Divide and Conquer ist also recht schnell - wesentlich besser als die vorher betrachteten Methoden.

Ein Nachteil des Verfahrens ist seine Anfälligkeit für Rechenungenauigkeiten, da durch die rekursive Teilung der Punktemenge viele längliche Bereiche mit fast parallel zu einander liegenden Kanten entstehen, bei deren Verarbeitung Probleme durch Rundungsfehler auftreten können.

9.2.5 Gewinnung aus dem Voronoi-Diagramm

Der Ansatz, eine Delaunay-Triangulierung mit Hilfe eines gegebenen Voronoi-Diagramms der betrachteten Punktemenge zu erzeugen, ist eigentlich kein echter Triangulierungsalgorithmus, da die erforderlichen Informationen bereits in Form des Voronoi-Diagramms vorliegen und lediglich transformiert werden. Die eigentliche Berechnungsarbeit muss bereits vorher durch die Erzeugung des Voronoi-Diagramms erbracht worden sein. Verfügt man jedoch aus irgend einem Grund bereits über das Voronoi-Diagramm einer Punktemenge, so kann man daraus, wie im Abschnitt über Voronoi-Diagramme bereits angesprochen, sehr einfach die dazugehörige Delaunay-Triangulierung gewinnen.

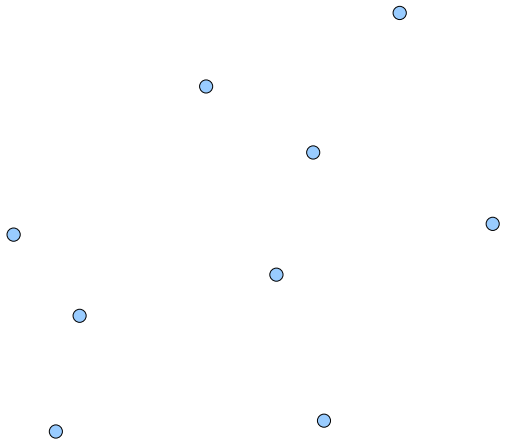
Wir haben im Abschnitt über das Voronoi-Diagramm gelernt, dass die Ecken des Voronoi-Diagramms gerade die Umkreismittelpunkte der Delaunay-Triangulierung sind. Daraus ergibt sich, dass wir die Delaunay-Triangulierung zu einer Punktemenge ganz einfach erhalten, indem wir um alle Ecke der Zellen ihres Voronoi-Diagramms Kreise „wachsen“ lassen und jeweils diejenigen Punkte zu einem Dreieck verbinden, die zuerst von einem Kreis berührt werden.

9.2.6 Beispiel: Der „Edge Flipping“-Algorithmus

Zur Erinnerung: Die Idee von „Edge Flipping“ war es, für eine beliebige gegebene Triangulierung jedes Paar von Dreiecken mit einer gemeinsamen Kante darauf zu überprüfen, ob es

die Umkreisbedingung erfüllt, und diese gegebenenfalls durch Kantentausch herzustellen. Im Folgenden wird diese Vorgehensweise an einem einfachen Beispiels veranschaulicht:

Die zu triangulierende Punktemenge:

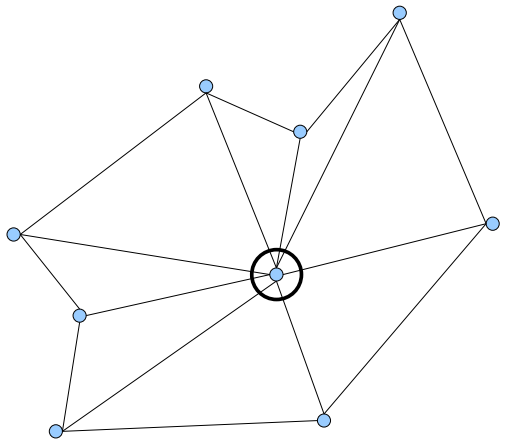


Schritt 1: Herstellung einer vorläufigen Triangulierung

- Finde den am mittigsten liegenden Punkt
- Sortiere die übrigen Punkte nach ihrer Winkelposition im Uhrzeigersinn um den gewählten Mittelpunkt
- Durchlaufe die Punkte in dieser Reihenfolge und verbinde jeden Punkt mit

  - dem **Mittelpunkt** und
  - dem **zuvor besuchten Punkt**

- Fertig, wenn man wieder am Ausgangspunkt angelangt ist



Schritt 2: Ausfüllen der konvexen Hülle

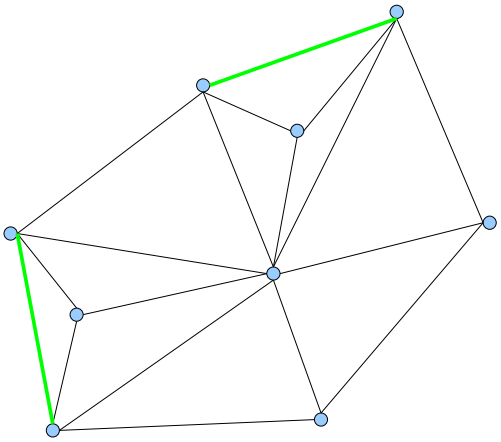
Problem:

Die Triangulierung deckt noch nicht die vollständige konvexe Hülle ab.

Lösung:

Umlaufe die Randpunkte erneut, finde „Einbuchtungen“ und „stopfe“ sie durch Verbindung ihrer beiden äußeren Eckpunkte.

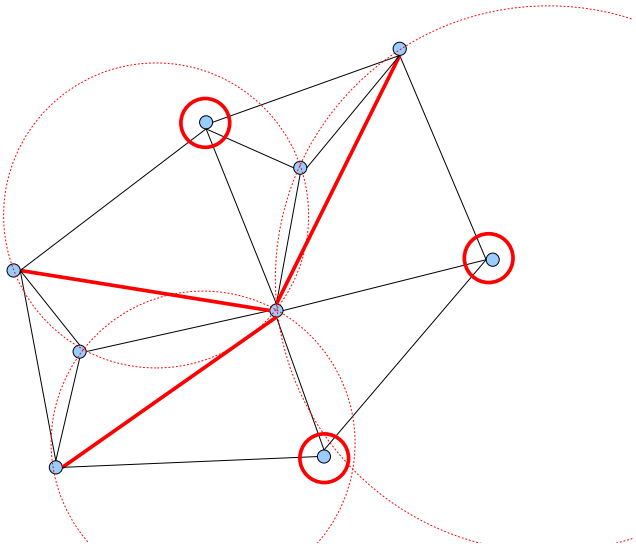
Bis zur vollständigen Abdeckung der konvexen Hülle muss dieser Schritt u.U. mehrfach wiederholt werden



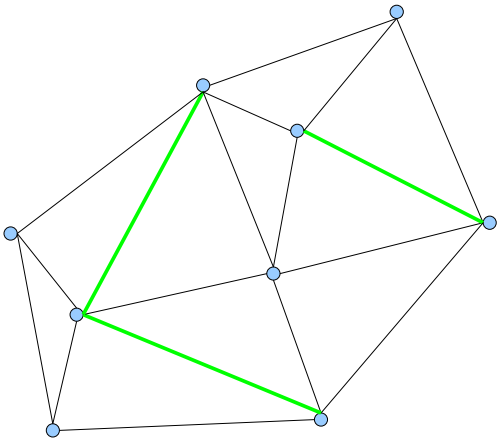
Schritt 3: Überprüfen und ggf. Herstellen der (Umkreisbedingung)

- Überprüfe alle Paare von aneinander grenzenden Dreiecken auf lokale Delaunay-Eigenschaft (**Umkreisbedingung**)
- Stelle diese ggf. durch „**Edge Flipping**“ her

Auch dieser Schritt muss evtl. mehrfach wiederholt werden, da es sein kann, dass ein Kantentausch die lokale Delaunay-Eigenschaft an anderer Stelle wieder zerstört.



Die fertige Delaunay-Triangulierung:



9.3 Constrained Delaunay-Triangulierungen

Manchmal soll eine Triangulierung **bestimmte Kanten unbedingt enthalten**, um wichtige Details im 3D-Modell zu erhalten (z.B. eine Straße durch eine Landschaft) - auch wenn dadurch **nicht mehr überall die Umkreisbedingung erfüllt ist**.

Darauf hat man hat jedoch bei der „reinen“ Delunay-Triangulierung keinen Einfluss. Ob eine bestimmte Kante letztlich in der Triangulierung enthalten ist, entscheidet allein der Triangulierungsalgorithmus, der aber nur darauf achtet, überall die Umkreisbedingung einzuhalten.

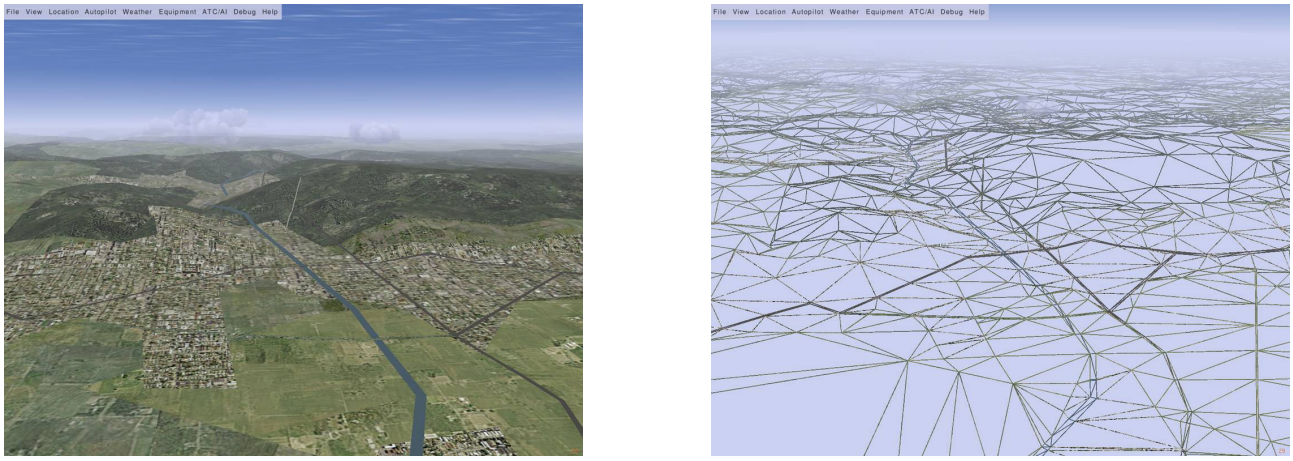


Abbildung 9.8: Die Kanten von strukturellen Besonderheiten wie Straßen und Flüsse sollen „eingeprägt“ werden, d.h. in der Triangulierung auf jeden Fall enthalten sein.

Für solche Fälle brauchen wird also eine neue Triangulierungs-Regel, die **Ausnahmen bei der Umkreisbedingung** zulässt, ansonsten aber **möglichst nah an der Delaunay-Triangulierung** bleibt. Dazu führen wir die Möglichkeit ein, bestimmte Kanten in die Triangulierung „einzuprägen“, die dann als „**undurchsichtig**“ gelten. Das bedeutet:

Definition: Ein Punkt heißt für einen anderen Punkt **nicht sichtbar**, wenn ihre Verbindungslinie eine eingeprägte, undurchsichtige Kante kreuzt.

Definition: Eine Kante zwischen zwei Punkten P1 und P2 einer Triangulierung T heißt **Constrained-Delaunay-Kante** genau dann, wenn ein Kreis K existiert, der P1 und P2 einschließt und höchstens andere Punkte aus S enthält, die zumindest von einem der Kantenendpunkte P1 oder P2 aus nicht sichtbar sind.

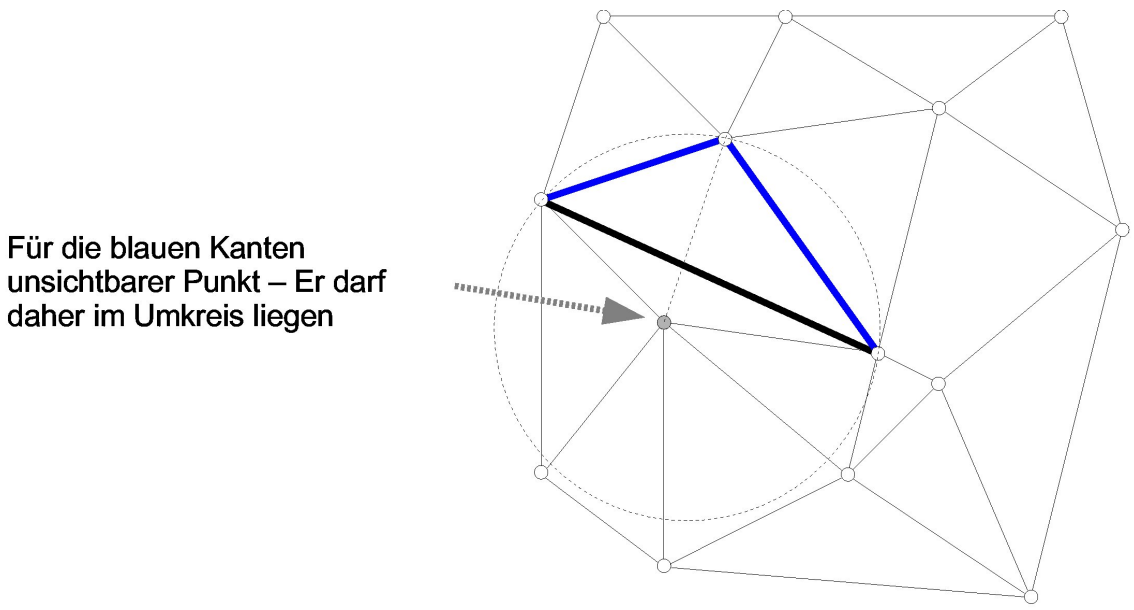


Abbildung 9.9: Constrained-Delaunay-Triangulierung

Definition: Eine Triangulierung heißt **Constrained-Delaunay-Triangulierung** genau dann, wenn jede ihrer Kanten eine **Constrained-Delaunay-Kante** ist bzw. im Umkreis jedes Dreiecks höchstens Punkte liegen, die für das jeweilige Dreieck **nicht sichtbar** sind.

9.4 Zusammenfassung

- **Delaunay-Triangulierungen** dienen zur Verbindung von Punkten im Raum zu massiven 3D-Strukturen

- Die **Umkreisbedingung** minimiert die Abweichung der Form vom Original.
- Das **Voronoi-Diagramm** ist der **duale Graph** zur Delaunay-Triangulierung: Es enthält die gleiche Information, stellt sie lediglich anders dar.
- Es gibt zahlreiche Algorithmen zur Erzeugung von Delaunay-Triangulierungen. Einer der einfachsten ist das **Edge Flipping**, ein typischer mit höherer Effizienz ist **Divide and Conquer**.
- **Constrained-Delaunay-Triangulierungen** bieten mehr Kontrolle über das Endergebnis durch Markierung von Kanten, die unbedingt enthalten sein sollen. Dabei sind sie immer noch „so delaunay wie möglich“.

Kapitel 10

Terrainvisualisierung

10.1 Motivation - Was ist Terrainvisualisierung?

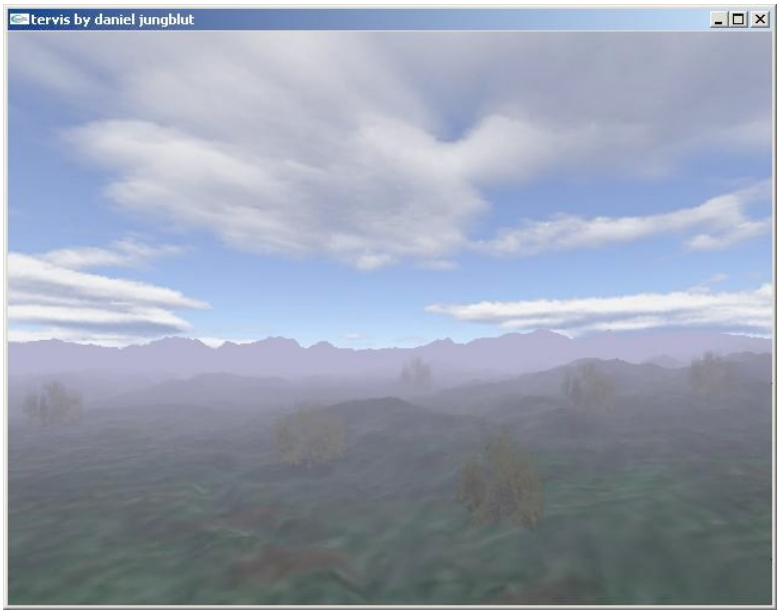


Abbildung 10.1: Beispiellandschaft in *OpenGL*

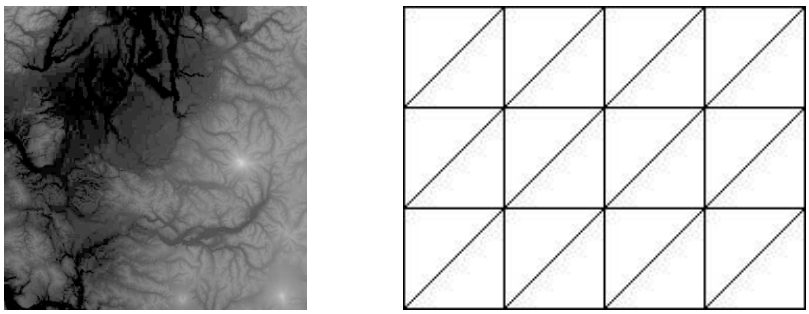
Im obigen Beispielsbild sind folgende Aspekte zu erkennen:

1. Eine mit *OpenGL* dargestellte Landschaft
2. Himmel
3. Bäume
4. Nebel

Im Folgenden wird erläutert wie diese Elemente dargestellt werden können. Inbesondere wird darauf eingegangen wie eine Lanschaft schnell dargestellt werden kann.

10.2 Heightmapbasierte Terraindarstellung

Zunächst wird eine Möglichkeit vorgestellt, wie man eine durch eine Heightmap gegebene Landschaft mit Hilfe von *OpenGL* darstellen kann.



(a) (b)

Abbildung 10.2: a) Beispiel für eine Heightmap b) Triangulierung

Jeder Pixel der Heightmap entspricht einem Vertex in dem Objektraum der Landschaft. Der Grauwert des Pixels gibt hierbei die Höehe der Lanschaft im entsprechenden Punkt an. Um das Terrain mit Hilfe von *OpenGL* zu visualisieren muss eine sinnvolle Triangulierung gewählt werden. Hierfür verbindet man die Pixel der Heightmap zunächst zu Quadraten, die dann in Dreicke unterteilt werden. Durch die Umkreisbedingung kann man leicht einsehen, dass die Triangulierung das Delaunaykriterium erfüllt. Um die Landschaft mit dem *OpenGL* Lichtmodell darstellen zu können, muss in jedem Vertex eine Normale bekannt sein. Die Normale in jedem Vertex lässt sich folgendermaßen berechnen:

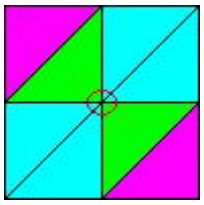


Abbildung 10.3: Gewichtung der Dreiecke bei der Normalenberechnung

Zunächst werden die Normalen der umliegenden Dreiecke berechnet.Diese werden mit folgender Gewichtung zur Normalen im rotumrandeten Vertex zusammengesetzt: Die cyanfarbenen Dreicke erhalten Gewicht $\frac{1}{8}$, die grünen Dreiecke erhalten Gewicht $\frac{1}{4}$. Die magentafarbenen Dreicke beeinflussen die Normale des mittleren Vertex nicht. Da die Normalenberechnung bei großen Heightmaps viel Zeit in Anspruch nimmt, ist es sinnvoll die Normaleninformation mit

in der Heightmap abzuspeichern, wofür der Grün- und der Blaukanal der Heightmap verwendet werden kann. Die Höheninformation wird dann im Rotkanal abgelegt. Dieses sogenannte *Smoothshading* ist notwendig, damit die Landschaft schön rund aussieht. Würde man die Normale nur für jedes Dreieck berechnen, würde an den Übergängen der Dreiecken Schattenkanten enstehen. Letzteres wird als *Flatshading* bezeichnet.

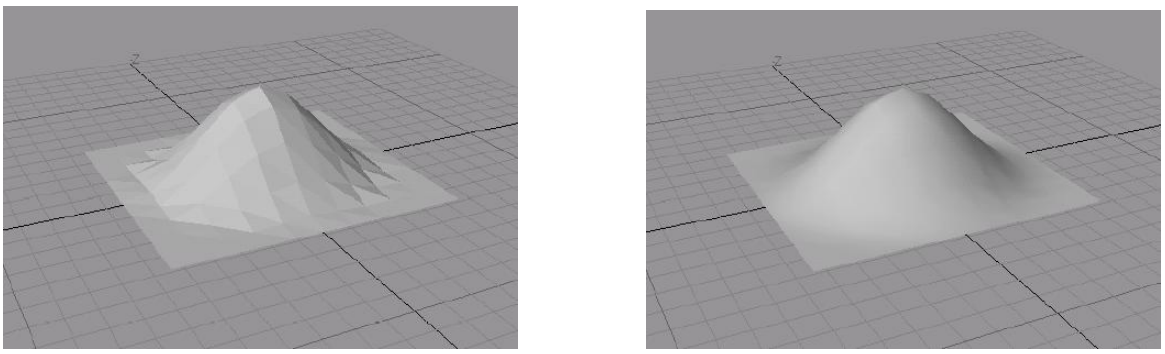


Abbildung 10.4: a) Flatshading b) Smoothshading

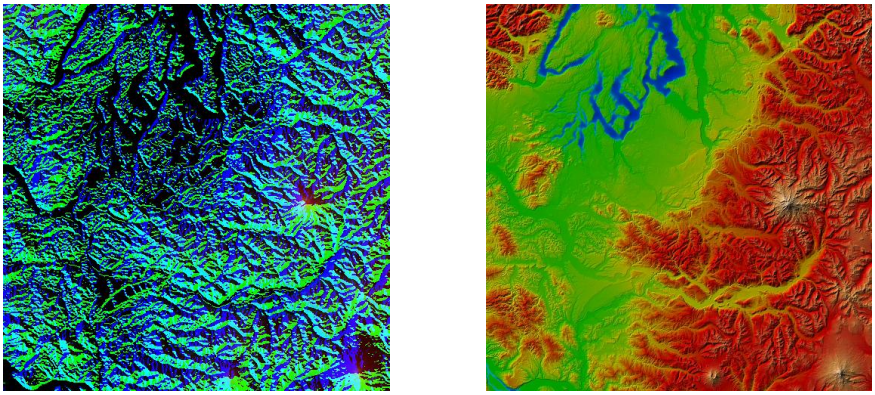


Abbildung 10.5: a) Heightmap mit Normaleninformation b) Verwendete Textur

10.3 Primitive Skybox

Um einen einfachen Himmel zu erstellen, wird ein großer Würfel um das Terrain gezeichnet. Dieser Würfel wird mit einer Himmelstextur versehen und unter Abschaltung des Lichtmodells gerendert. Die Skybox bewegt sich immer mit dem Betrachter mit, so dass sich die Kamera immer in der Mitte der Skybox befindet.

Eine solche Skybox erzeugt allerdings nur realistische Eindrücke, wenn der Betrachter sich langsam bewegt. Für einen Flugsimulator beispielsweise wäre diese Art Skybox nicht geeignet. Zudem ist die Wolkenerscheinung statisch, da sich die Wolken nicht bewegen.

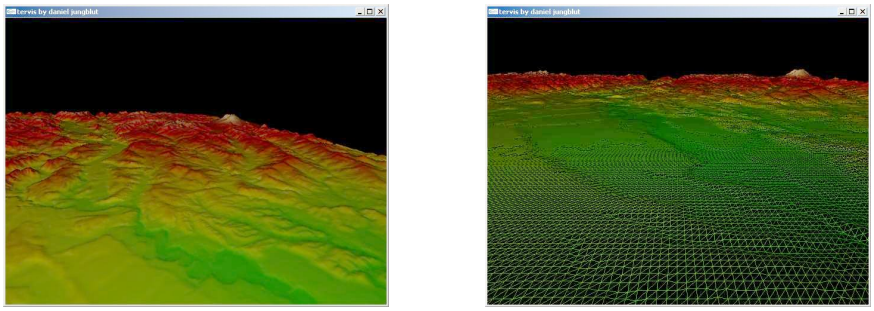


Abbildung 10.6: a) Mit *OpenGL* gerendertes Terrain b) Drahtgittermodell des Terrains

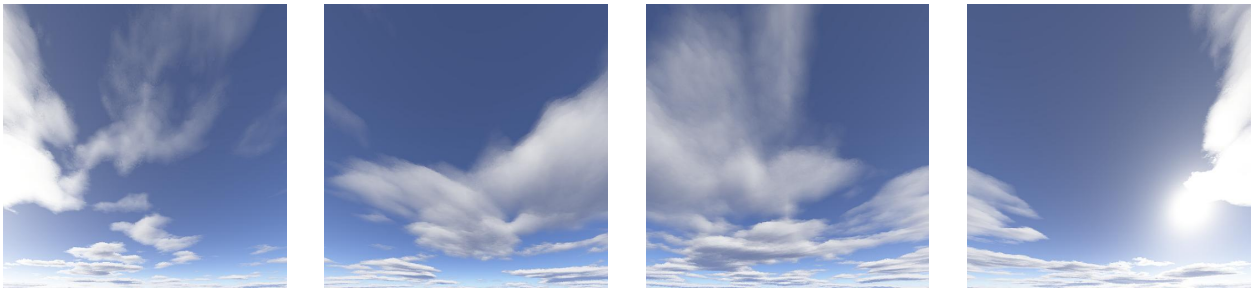


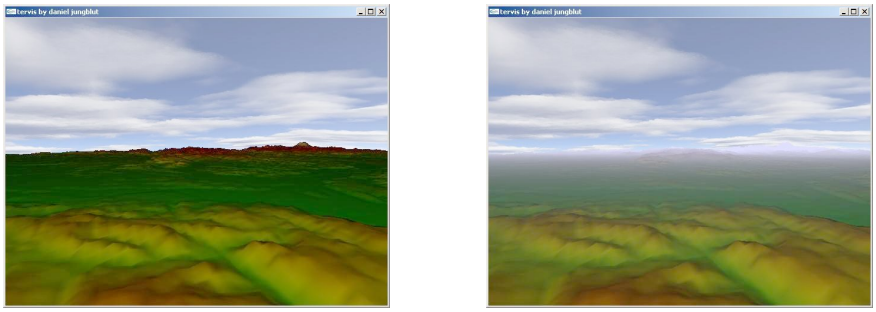
Abbildung 10.7: Texturen für Innenseite der Skybox

Abbildung 10.8: Ergebnis der primitiven Skybox

10.4 Nebel in OpenGL

Nebel ist bereits in *OpenGL* integriert und wird von der Graphikkarte berechnet. Es können atmosphärische Effekte erzielt werden und unter Umständen kann die Rendergeschwindigkeit erhöht werden. Um letzters zu erreichen sollte eine lineare Nebelfunktion verwendet werden. Hierbei wird ein Startwert S und ein Endwert E für den Nebel angegeben. Zudem wird dem

Nebel eine Farbe zugewiesen Wenn ein Punkt im Objektraum von der Kamera entfernt ist als E verschwindet er völlig im Nebel. Ist der Punkt näher an der Kamera als S wird er nicht mit Nebel überdeckt. Dazwischen wird der Nebel linear interpoliert über die Objekte gelegt. Um den Abstand von Kamera zu einem Vertex zu erhalten wird der Wert aus dem Z-Buffer verwendet.



(a) (b)

Abbildung 10.9: Lanschaft ohne Nebel (a) und mit Nebel (b)

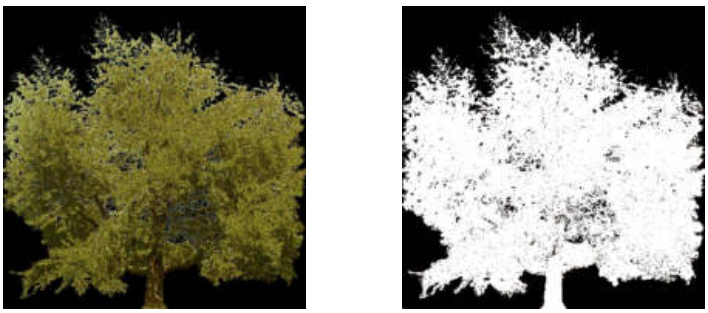
Wie in Abbildung 9 zu sehen kann durch Nebel der Aliasingeffekt am Horizont reduziert werden. Durch die Einschränkung der Sichtweite kann zudem die Rendergeschwindigkeit erhöht werden, da die Clipping Plane entsprechend dem Endwert für den Nebel verkleinert werden kann und so weniger Dreiecke gerendert werden müssen. Diese Technik findet beispielsweise im Echtzeitrollenspiel *Gothic 2* Anwendung. Hier kann die Sichtweite dynamisch der Rechnergeschwindigkeit angepasst werden.

Abbildung 10.10: Nebel in *Gothic 2*

10.5 Bäume aus vier Dreicken

Einfache Bäume kann man erzeugen, indem man zwei Vierecke, also vier Dreiecke, überkeuzt und mit einer entsprechenden Textur versieht. Hierbei werden Teile der Textur mit Hilfe des Al-

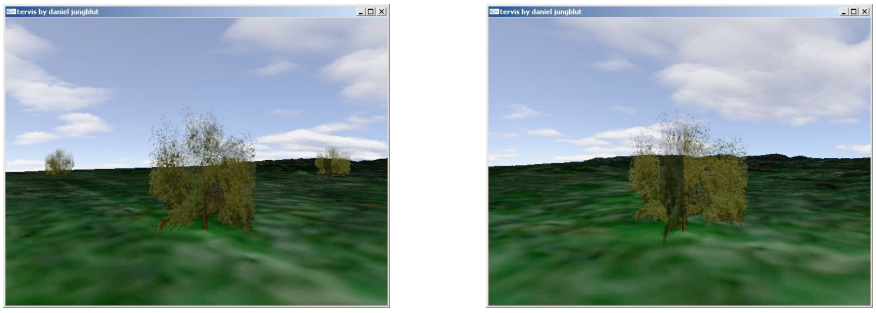
phakanals durchsichtig gezeichnet. Wichtig ist hierbei die Bäume, wie auch andere transparente Flächen ganz am Ende des Rendervorgangs zu zeichnen.



(a) (b)

Abbildung 10.11: a) Baumtextur b) Zugehöriger Alphakanal

Die in Abbildung 11 schwarzen Teile der Textur bzw. des Alphakanals sind hierbei durchsichtig.



(a) (b)

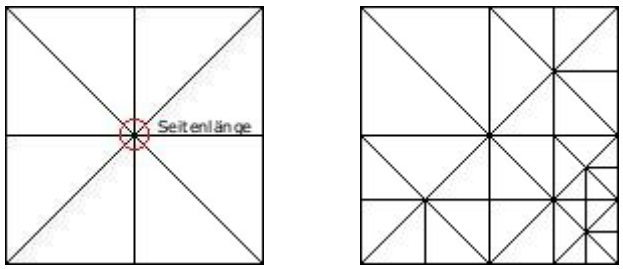
Abbildung 10.12: Verschiedene Ansichten eines primitiven Baums

In Abbildung 12a ist ein einfacher Baum zu sehen, der durch den Blickwinkel relativ realistisch erscheint. Dreht man die Kamera jedoch sieht man die Kanten des Baums (Abbildung 12b), da dieser lediglich aus zwei überkreuzten Vierecken besteht. Im Echtzeitrollenspiel *Gothic 2* wird die Technik des Alphablendings ebenfalls im Zusammenhang mit Bäumen und Büschen verwendet, wobei die Objekte hier aus mehr Dreiecken zusammengesetzt sind.

10.6 SOAR-Algorithmus

Bisher wurde die Triangulierung statisch berechnet. Dadurch werden unabhängig von der aktuellen Kameraposition immer gleich viele Dreiecke gerendert. Es wäre aber beispielsweise sinnvoll weit entfernte Objekte mit wenigen Dreiecken zu zeichnen, oder Objekte die sich außerhalb des Sichtkegels befinden gar nicht zu rendern. Hierzu muss die Triangulierung des Terrains in jedem

Abbildung 10.13: Baum in *Gothic 2*



(a) (b)

Abbildung 10.14: Triangulierung (a), die dynamische Verfeinerungen zuläßt (b)

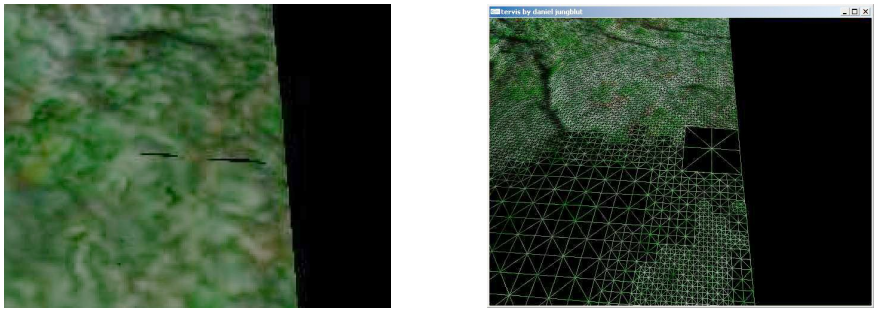
Bild neu berechnet werden. Zunächst benötigt man eine Triangulierung, die diese dynamischen Änderungen zuläßt.

Der SOAR Algorithmus verfolgt bei der dynamischen Triangulierungsberechnung folgende Idee: Weit entfernte und ebene Flächen sollen mit wenigen Dreiecken dargestellt werden. Die Triangulierung wird also in Abhängigkeit des Aufenthaltsorts der Kamera berechnet.

Um eine Triangulierung wie in Abbildung 14a zu zeichnen, benötigt man den Mittelpunkt (rot umrandet) und die zugehörige Seitenlänge. Möchte man von einem Punkt aus keine Triangulierung zeichnen sondern noch weiter verfeiern, so sind die neuen Knoten jeweils eine halbe Seitenlänge rechts bzw. links und oberhalb bzw. unterhalb des aktuellen Knotens. Dies kann durch Rekursion relativ einfach gelöst werden, in man rekursiv einen Quadtree absteigt, bis man auf der feinsten Gitterebene angekommen ist, von der aus man immer Triangulierungen zeichnet.

Wie in Abbildung 15 zu erkennen ist, dürfen sich benachbarte Knoten nur um eine Detailstufe unterscheiden, da sonst Löcher im Mesh enstehen können. Dies muss bei der dynamischen Berechnung der Triangulierung beachtet werden.

Berechnung der dynamischen Triangulierung: Zunächst muss in jedem Knoten ein Fehler berechnet werden, der angibt wie groß der Unterschied zwischen einer groben zu einer feineren



(a) (b)

Abbildung 10.15: a) Loch im Mesh, b) Ursache des Lochs

Triangulierung ist. Hierzu wird in jedem Knoten der maximale Höhenunterschied zwischen der groben und der nächstfeineren Unterteilung ist. Ist dieser Fehler zu einem benachbarten Knoten zu groß wird das Maximum der beiden Fehler für beide Knoten verwendet. Hierdurch werden Löcher im Mesh vermieden. Die Fehlerberechnung wird zu Programmstart einmal ausgeführt und die Fehler werden abgespeichert. Vor jedem Bild, das gezeichnet wird, werden die einzelnen Knoten in der oben beschriebenen Quadtreestruktur durchlaufen. In jedem Knoten wird dann entschieden ob von ihm aus eine Triangulierung gezeichnet wird oder ob von diesem Knoten aus weiter verfeinert werden soll. Diese Frage wird durch folgende Formel beantwortet:

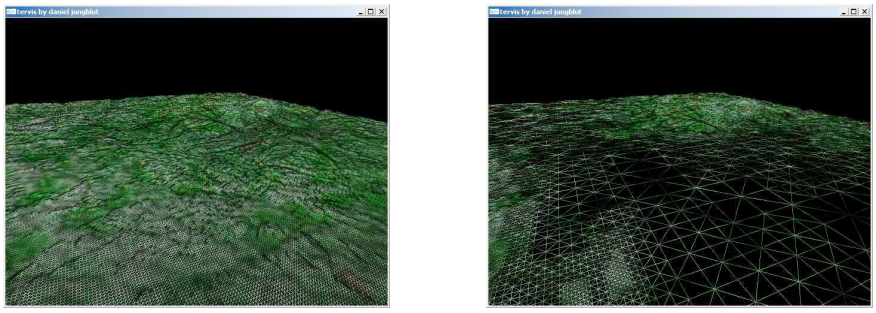
$f = \frac{size * C * max\{c * fehler, 1\}}{I}$

$f \geq 1 \Rightarrow$ weiter unterteilen

$f < 1 \Rightarrow$ Triangulierung von aktuellem Knoten aus zeichnen

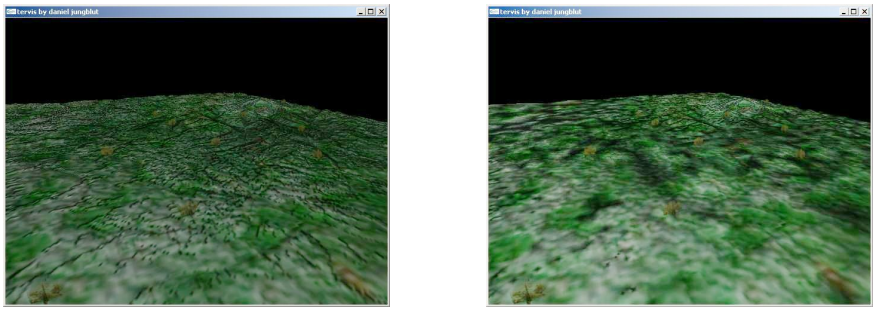
Hierbei ist $size$ die Seitenlänge des aktuellen Knotens, $fehler$ der berechnete Fehler im aktuellen Knoten und I der Abstand der Kamera vom aktuellen Knoten. C und c sind Parameter die angepaßt werden können. C gibt allgemein an, wie hoch die Landschaft aufgelöst ist, c gibt an wie hoch die Auflösung in Bereichen mit großem Oberflächenfehler ist. Das Maximum über $\{c * fehler, 1\}$ wird gebildet, dass unabhänging davon wie klein ein Fehler ist in einem Knoten ist, immer die höchst mögliche Auflösung verwendet wird, wenn sich die Kamera nah genug an dem Knoten befindet.

In Abbildung 17 ist ein relativ großer Unterschied zwischen dem Terrain mit und ohne SOAR zu erkennen. Dieser Unterschied wird wesentlich kleiner, wenn sich die Kamera nur knapp über der Terrainoberfläche bewegt, beispielsweise in einem 3D Rollenspiel oder einem Shooter. Der Detailverlust stammt hauptsächlich daher, dass bei großen Dreiecken weniger Normalen zur Lichtberechnung des Bildes miteinbezogen werden.



(a) (b)

Abbildung 10.16: Triangulierung a) ohne SOAR, b) mit SOAR



(a) (b)

Abbildung 10.17: Terrain a) ohne SOAR, b) mit SOAR

10.7 Erweiterungen des SOAR-Algorithmus

Der SOAR Algorithmus berechnet eine Triangulierung in Abhängigkeit der Position der Kamera. Es ist aber auch sinnvoll die Dreiecke die sich außerhalb des Sichtkegels der Kamera befinden nicht zu zeichnen. Hier kommt das *Frustum-Culling* ins Spiel. Mit Hilfe der Projection- und der Modelviewmatrix, die von der der Graphikkarte ausgelesen werden können, wird beim Durchlaufen des Quadtrees der Abstand zwischen aktuellem Knoten und Sichtkegel der Kamera bestimmt. Ist dieser größer als die aktuelle Seitenlänge multipliziert mit $\sqrt{2}$, so liegen alle Punkte die von dem Knoten ausgehend gerendert werden könnten außerhalb des Sichtkegels. Somit wird dieser Knoten und alle im Quadtree weiter unten liegende Knoten im aktuellen Bild ignoriert.

Beim Bewegen der Kamera kann sich das Gelände teilweise sprunghaft ändern, wenn sich an einer Stelle die Triangulierung zum vorherigen Bild ändert. Dieses Phänomen wird als *Popping* bezeichnet. *Popping* tritt auch in vielen Computerspielen auf, wenn beispielsweise Objekte plötzlich auftauchen oder verschwinden. In Kombination mit dem SOAR-Algorithmus kann dies durch *Geomorphing* gelöst werden. Wenn sich an einer Stelle die Verfeinerung der Triangulierung zum vorherigen Bild erhöht, erfolgt diese Änderung nicht instantan. Vielmehr wird das Gelände genau so gezeichnet wie im vorherigen Bild, allerdings werden hierzu mehr Drei-

ecke verwendet, eben soviele wie die feinere Triangulierung vorsieht. In den nächsten Bildern wird dann das Terrain an der entsprechende Stelle so verändert bis nach einer gewissen Zeit das Gelände so erscheint, wie es unter der feineren Triangulierung aussehen soll. Hierbei kann beispielsweise eine lineare Interpolation für die Höhe des Terrains verwendet werden. Wird die Detailstufe des Terrains verringert, wird zunächst die aktuelle Triangulierung beibehalten, die Höhe des Geländes aber so lange verändert, bis das gezeichnete Gelände mit dem gröber triangulierten Gelände übereinstimmt. Anschließend wird die Triangulierung vergröbert. Durch das plötzliche Ändern der Triangulierung kommt es dennoch zu einem Poppingeffekt, wenn instantan die zum Rendern verwendete Normaleninformation erhöht oder verringert wird, indem feinere Dreiecke hinzukommen oder weggelassen werden. Dies kann mit einer sogenannten *Normalmap* behoben werden. Durch einen entsprechenden *Shader* kann so immer eine konstante Normaleninformation in das Bild einfließen, unabhänging davon wieviele Dreiecke gerendet werden. Diese Technik kann allerdings erst auf neueren, *OpenGL 2.0* fähigen Graphikkarten angewandt werden.

10.8 Ausblick

Der SOAR Algorithmus kann weiter beschleunigt werden, in dem die oft durchlaufenen Quoten nahe beisammen im Speicher abgelegt werden. Mehrere Ansätze hierfür werden im folgenden Paper ausgiebig diskutiert:

http://www-static.cc.gatech.edu/̃lindstro/papers/vis2001a/

Weitere interessante Algorithmen zum Thema *Level of Detail* und Terrainvisualisierung können hier gefunden werden:

http://vterrain.org/LOD/Papers/index.html

10.9 Quellen

http://wiki.delphigl.com/index.php/Tutorial_Alphamasking

http://wiki.delphigl.com/index.php/Tutorial_Terrain1

http://wiki.delphigl.com/index.php/Tutorial_Terrain2

http://wiki.delphigl.com/index.php/Tutorial_Terrain3

http://www-static.cc.gatech.edu/̃lindstro/software/soar/

Literaturverzeichnis

[1] S. Matsuba und B. Roehl: VRML. Das Kompendium, München 1996

[2] O. Schlüter: VRML. Sprachmerkmale, Anwendungen, Perspektiven, Köln 1998

[3] K. Zeppenfeld: Lehrbuch der Grafikprogrammierung. Grundlagen, Programmierung, Anwendung, München 2004

[4] http://www.secondlife.com/whatis/

[5] http://www.uni-kassel.de/hrz/anwendungen/matthias/hrz-info/vrml/vrml.html

[6] http://de.wikipedia.org/wiki/Hauptseite

[7] http://www.ztt.fh-worms.de/de/others/sem/ws97_98/vrml/c30.html

[8] http://www.vlc.com.au/~justin/opinion/goodbye_vrml.html

[9] http://www.vrmlsite.com/sep96/spotlight/top/top.html

[10] http://home.snafu.de/hg/worlds.htm

[11] http://www.parallelgraphics.com/products/

[12] http://seminare.snm-hgkz.ch/~maja/seminare/ideol2.html

[13] Grady Booch. *Objektorientierte Analyse und Design*, Addison-Wesley GmbH, 1994, ISBN 3-89319-673-0

[14] Nicolai Josuttis. *Objektorientiertes Programmieren in C++*, Addison-Wesley GmbH, 1994, ISBN 3-89319-637-4

[15] Ralf Kühnel. *Die Java-Fibel*, Addison-Wesley GmbH, 1994, ISBN 3-8273-1024-5

[16] Stanley B. Lippman *C++ lernen und beherrschen*, Addison-Wesley GmbH, 1994, ISBN 3-89319-375-8

[17] David Flanagan. *Java in a Nutshell (dt. Ausgabe)*, O'Reilly & Associates, Inc., 1996, ISBN 3-930673-46-0

[18] Klaus Matzdorff. *Objektorientierte Softwareentwicklung mit C++*, Steuer- und Wirtschaftsverlag Hamburg, 1994, ISBN 3-89161-251-6

[19] Pierre-Alain Muller. *instant UML* Wrox Press Ltd., 1997, ISBN 1-861000-87-1

[20] Steffen Schäfer. *Objektorientierte Entwurfsmethoden*, Addison-Wesley GmbH, 1994, ISBN 3-89319-692-7

[21] Robert Sedgewick. *Algorithmen in C++*, Addison-Wesley GmbH, 1992, ISBN 3-89319-462-2

[22] Bjarne Stroustrup. *Die C++ Programmiersprache*, Addison-Wesley GmbH, 2.Auflage 1994, ISBN 3-89319-386-3

[23] Bender, Michael ; Brill, Manfred: *Computergrafik*. Hanser, 2003. – ISBN 3–446–22150–6

[24] Brüderlin, Beat ; Meier, Andreas: *Computergrafik und Geometrisches Modellieren*. Teubner, 2001. – ISBN 3–519–02948–0

[25] Bloomenthal, Jules ; Rokne, Jon: Homogeneous Coordinates. In: *The Visual Computer* 11 (1994), January, Nr. 1, S. 15–26. `http://dx.doi.org/10.1007/BF01900696`. – DOI 10.1007/BF01900696

[26] Fellner, W. D.: *Computergrafik*. B.I.-Wissenschaftsverlag, 1992. – ISBN 3–411–15122

[27] William T. Reeves: Particle Systems – A Technique for Modeling a Class of Fuzzy Objects (1983) 56

[28] Andrew Witkin: Physically Based Modelling: Principles and Practice. Particle System Dynamics (1997) 59