

Fachhochschule Wedel

Studiengang Medieninformatik

Eine Einführung in die prozedurale Landschaftsgenerierung

Seminararbeit

Tjark Smalla
Matrikel-Nummer 100554

Betreuer Prof. Bohn

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
1 Einleitung	1
1.1 Implizite vs. explizite Funktionen	2
2 Diamon-Square Algorithmus	3
3 Fourier-Spektralsynthese	5
4 Noise	6
4.1 Grundlagen	6
4.1.1 Lattice-Function	6
4.1.2 Interpolation und Fade-Function	7
4.2 Value-Noise	8
4.3 Gradient-Noise	9
4.4 Fractal-Noise	9
4.5 Ausblick: Simplex-Noise	10
4.5.1 jetzt geht es noch tiefer	11
A Anhang	13
A.1 Quelltexte	13
Literaturverzeichnis	16

Abbildungsverzeichnis

2.0.1 2 Rekursionschritte des Diamond Square Algorithmus bei einem quadratischen Höhenfeld mit der Auflösung 5x5	4
4.1.1 Fade-Function	8
4.5.1 Test-Bild	11
4.5.2 Zwei Bilder werden mit dem \LaTeX -Paket subcaption nebeneinander angezeigt	12

Tabellenverzeichnis

4.1	eine sinnlose Tabelle	10
-----	---------------------------------	----

1 Einleitung

Die synthetische Erzeugung von Höhendaten zur Darstellung von möglichst realistischen Landschaften fand insbesondere nach der Vorstellung einer Rauschfunktion¹ von Ken Perlin 1985[K85] viel Aufmerksamkeit. Zwar gab es in den letzten Jahren nur noch wenige Veröffentlichungen zu dem Thema, allerdings kam ein neues Feld auf in dem die prozedurale Erzeugung von natürlichen Strukturen eine große Rolle spielt - die Computerspiele. Als bekanntestes Beispiel ist hier sicherlich das Spiel Minecraft vom Studio Mojang zu nennen, welches einen Großteil seiner Faszination aus der komplett prozedural erzeugten veränderbaren Landschaft zieht.

Im folgenden werden 3 bewährte Algorithmen für diesen Zweck vorstellen und darauf eingegangen, für welchen Zweck sie besonders geeignet sind. In Hinblick auf das Einsatzgebiet von Computerspielen sollen sie dabei insbesondere auf die Möglichkeit zur effektiven Speicherung von sehr großen Landschaften untersucht werden. Zuerst wird der Diamond-Square Algorithmus[FFC82] vorgestellt, welchen man verallgemeinert auch als einen *Polygon unterteilungs Algorithmus* bezeichnen kann. Danach wird kurz die Spektalsynthese mit der Fourier-Methode erläutert bevor es eine kurze Einführung in die Welt der Rauschfunktionen gibt. Neben der Synthese von Landschaften sind diese Algorithmen vielseitig einsetzbar. Insbesondere bei der bereits erwähnten Kategorie der Rauschfunktionen ist davon auszugehen, dass sie auch in proprietärer 3D-Software sowie Computerspielen immer noch als wichtiger Algorithmus genutzt wird. Dies liegt vor allem an ihrer einfachen Anpassung sowie Skalierbarkeit in mehreren Dimensionen durch die auch Wolken, Feuer, Rauchverwirbelung und sogar Fellverteilung dargestellt werden kann[DSE03].

1 Auch bekannt als *Perlin-Noise*

1.1 Implizite vs. explizite Funktionen

Alle hier vorgestellten Methoden lassen sich in 2 Gruppen einteilen: implizite und explizite Funktionen. Während eine explizite Funktion alle Höhenpunkte auf einmal berechnet lässt sich die implizite Funktion für jeden Punkt, also jede Koordinate, isoliert auswerten.

Durch die Unabhängigkeit der Berechnung für jeden einzelnen Punkt lassen sich implizite Algorithmen sehr effizient parallel auf einer GPU berechnen². Dies ermöglicht die Ausführung zur Laufzeit, während explizite Methoden in der Regel vor oder bei Programmstart einmalig berechnet werden und deren Ergebnisse in einer Textur gespeichert werden. Dies hat den Vorteil, dass der Speicherbedarf teilweise enorm sinken kann.

Ein Einsatzgebiet für diese Technik ist das Bump-Mapping bzw. Displacement Mapping bei der zusätzliche Höhenwerte auf ein Objekt durch Shading oder neue Vertices auf der Objektoberfläche hinzugefügt werden[Rus]. Da moderne Echtzeitspiele immer mehr und immer größere Texturen verwenden steigt der Bedarf an Speicher enorm wenn für jede Textur noch eine Normal/Bump/Displacement Map gespeichert werden muss. Implizite Methoden erlauben es, anstatt der Texturen einige Parameter in Form von Floats und Integern zu speichern. Auch eine Anpassung des Detailgrades ist zur Laufzeit ohne Probleme möglich, während bei der Detailgrad bei expliziten Methoden von der Auflösung der Textur abhängt³.

Der Diamond-Square Algorithmus sowie die Spektralsynthese gehören zu der Gruppe der expliziten Algorithmen, während die Rauschfunktionen implizit auswertbar sind.

² Siehe Beispielimplementierung in einem Vertex-Shader

³ Diese Eigenschaften lassen sich zwar auch durch die Berechnung von expliziten Methoden zur Laufzeit erreichen, jedoch lassen diese sich wie erwähnt nicht effektiv durch die GPU beschleunigen wodurch die Berechnung innerhalb eines Frames unperformant ist.

2 Diamon-Square Algorithmus

Die Vorteile des Diamond-Square Algorithmus liegen insbesondere in seinen, trotz seiner einfachen Implementierung, optisch plausiblen Ergebnissen. Der Algorithmus unterteilt sich pro Rekursionsschritt in 2 Phasen¹: der Diamond und der Square Step. Das zu berechnende Höhenfeld muss dabei in der Auflösung eine Breite bzw. Höhe von $2n + 1$ aufweisen.

Zur Initialisierung werden den vier Eckpunkte des zu berechnenden Höhenfeldes jeweils zufällige Höhenwerte zugewiesen. Außerdem wird eine von der Rekursionstiefe abhängige Offset-Funktion $O(k) = (rnd() * 2 - 1) / 2^k$ definiert, wobei $rnd()$ einen Zufallswert zwischen 0-1(inklusive) zurückgibt.

Im Diamond Step wird nun der Mittelpunkt des Höhenfeld gesucht und mittels einer beliebigen Interpolation² zwischen den 4 Höhenwerten der Eckpunkte ein Wert gefunden, welcher dann mit der Offset-Funktion zufällig verschoben wird. Bei dem Square Step werden nun die jeweiligen mittleren Randpunkte zwischen den bereits initialisierten Eckpunkten durch eine einfache Interpolation zwischen den 2 nächstliegenden

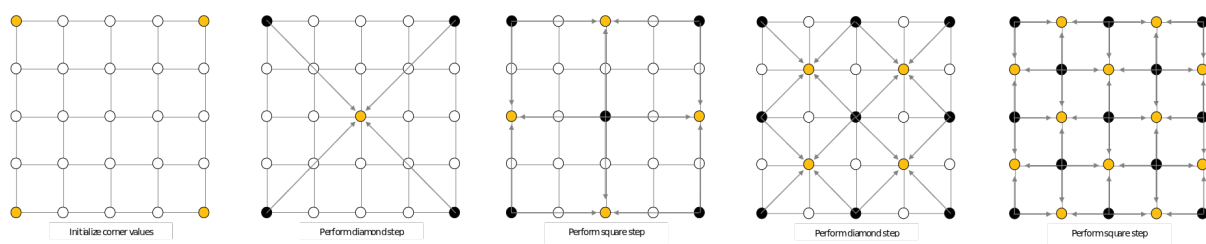


Abbildung 2.0.1: 2 Rekursionschritte des Diamond Square Algorithmus bei einem quadratischen Höhenfeld mit der Auflösung 5x5

¹ Auch *Steps* genannt

² In der Beispielimplementierung wurde eine bilineare Interpolation gewählt

2 Diamon-Square Algorithmus

Eckpunkten berechnet. Wie in Abbildung 2.0.1 zu sehen ergeben sich daraus 4 weitere Vierecke mit berechneten Eckpunkten, auf die der Algorithmus wieder angewandt wird.

3 Fourier-Spektralsynthese

4 Noise

Synthetisch erzeugtes Rauschen (*engl. Noise*) erweist sich als hilfreiches Mittel zur Erzeugung von zufällig erscheinenden Strukturen. Als wohl bekannteste Implementierung ist hier die Implementierung von Ken Perlin[K85] zur Erzeugung einer Marmortextur auf einer Vase zu nennen¹.

Neben umfangreichen Anpassungsmöglichkeiten durch verschiedene Parameter ist die Performance dieses Verfahrens ein entscheidender Grund für die Nutzung. Noise verbraucht extrem wenig Speicher, ist relativ einfach zu berechnen und ist zu jeder Zeit an einer beliebigen Stelle auswertbar, was es auch für Echtzeitanwendungen geeignet macht.[HH]

Dieses Kapitel soll ein grundlegendes Verständnis über Noise-Funktionen bieten. Dazu werden zuerst grundlegende Komponenten, welche jeder Implementierung zugrunde liegen, erläutert. Anschließend werden *Value-Noise* 4.2, *Gradient-Noise* 4.3 sowie *Fractal-Noise* 4.4 erklärt, bevor es einen Ausblick auf den *Simplex-Noise* 4.5 Algorithmus gibt.

Weitere ausführliche Beschreibung in [Bur08] (Gradient-Noise) und in [Gus05] (Simplex-Noise).

4.1 Grundlagen

4.1.1 Lattice-Function

Der erste Schritt zur Erzeugung von Noise ist in der Regel eine sogenannte *Noise-Lattice* (*Rausch-Gitter*)-Funktion[AC] der Form $l(\vec{k}) : Z^n \mapsto [-1, 1]$. Diese dient zur Beschreibung eines

¹ Auch als *Perlin-Noise* bezeichnete Implementierung von Gradient Noise in 3-D

4 Noise

Gitters, welches die Form unserer zukünftigen Noise-Funktion bestimmen wird. Die Funktion muss dabei unbedingt deterministisch sein².

Die Wahl der Lattice Funktion ist entscheidend für das spätere Erscheinungsbild der Noise-Funktion. Eine gleichverteilte Folge von pseudo zufällige Zahlen wie sie etwa die meisten in Programmiersprachen implementierten Zufallsgeneratoren bieten erfüllt zwar die Anforderungen der Deterministik der Funktion, kann allerdings zu unerwünscht starken Differenzen zwischen zwei Benachbarten Gitterpunkten führen. Im folgenden gehen wir von einer Standardnormalverteilung aus um die Wahrscheinlichkeit für Werte nahe den Intervalgrenzen zu verringern.

4.1.2 Interpolation und Fade-Function

Um aufbauend auf der Lattice-Funktion 4.1.1 eine Funktion $S(\vec{x}) : \mathbb{R}^n \mapsto \mathbb{R}, \vec{x} \in \mathbb{Z}^n$ zu definieren wird zwischen benachbarten Gitterpunkten lokal interpoliert. Dafür wird eine sogenannte Fade-Function [Per] der Form $f(t) : \mathbb{R} \mapsto \mathbb{R}$ mit $t \in [0, 1]$ definiert, welche den Übergang zwischen den Gitterpunkten steuert.

Um überhaupt eine stetige Noise-Funktion zu ermöglichen, muss

$$f(0) = 0 \wedge f(1) = 1 \quad (4.1)$$

gelten. Damit der Übergang zwischen den Gitterpunkten möglichst glatt und damit natürlich wirkt, sollte jedoch eine Stetigkeit von C^2 und damit die Eigenschaften

$$f'(0) = f'(1) = 0 = f''(0) = f''(1) \quad (4.2)$$

gelten.

Dafür wird im folgenden das Polynom $f(t) = 6t^5 - 15t^4 + 10t^3$ benutzt, welches auch in Perlins Referenzimplementierung Verwendung findet [Bur08] und alle Eigenschaften erfüllt.

Die mit $f(t)$ gebildete, interpolierende Funktion $S(\vec{x}) = \text{noise}(\vec{x})$ definiert nun - im einfachsten Fall - die Noise-Funktion.

² Siehe 4.4, sie muss also für jede Koordinate eines Gitterpunktes immer denselben Funktionswert liefern.

4 Noise

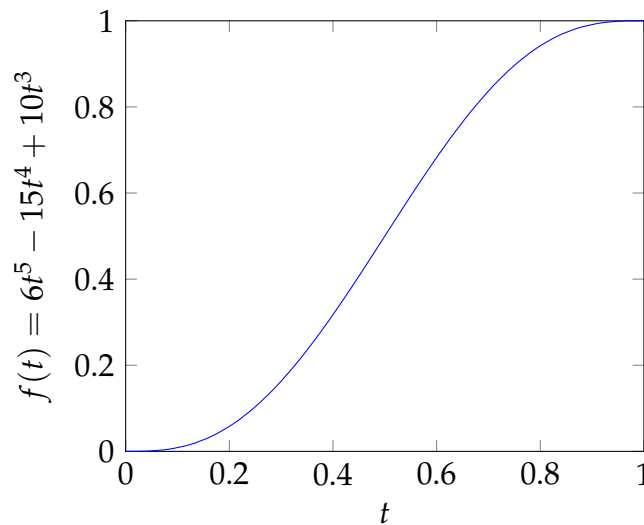


Abbildung 4.1.1: Fade-Funktion

4.2 Value-Noise

Value-Noise ist die wohl naivste Implementierung einer Noise-Funktion. Bei ihr werden die Gitterpunktwerte, welche durch die *Lattice-Funktion* 4.1.1 erzeugt wurden, als Höhenwerte interpretiert.

Untenstehend ist eine Implementierung in C# für eine 2-Dimensionale Rauschfunktion zu sehen. Noise lässt sich problemlos in mehrere Dimensionen skalieren. Einzig die Interpolation der Werte muss hier angepasst werden. In der Implementierung ist zu sehen, wie die Gitterpunktwerte - ähnlich einer *billinearen Interpolation* - mit der *Fade-Funktion* interpoliert werden.

Value-Noise Implementierung C#

```
1 public float S1(float x, float y) {
2     int floorX = Mathf.FloorToInt(x),
3     ceilX = Mathf.CeilToInt(x),
4     floorY = Mathf.FloorToInt(y),
5     ceilY = Mathf.CeilToInt(y);
6     float tx = x - floorX,
7     ty = y - floorY;
8
9     return FadeFunction(1 - ty) *
10        (LatticeFunc(floorX, floorY) * FadeFunction(1 - tx) + FadeFunction(tx) * LatticeFunc(ceilX, floorY))
11        +
12        FadeFunction(ty) *
13        (LatticeFunc(floorX, ceilY) * FadeFunction(1 - tx) + FadeFunction(tx) * LatticeFunc(ceilX, ceilY));
14 }
```

4.3 Gradient-Noise

Der vorher erwähnte Value-Noise kann, je nach Parameterwahl, noch ein unruhiges Rauschen erzeugen. Um die Übergänge zwischen den Gitterpunktwerten noch sanfter und damit natürlicher aussehen zu lassen wurde der Gradient-Noise erfunden. Hier werden die Gitterpunktwerte nicht als Höhenwerte, Gradienten an den Nullstellen der Noise-Funktion gesehen. Es ergibt sich also:

$$\text{noise}(\vec{k}) = 0 \wedge \text{noise}'(\vec{k}) = S(\vec{k}), k \in \mathbb{Z}^n. \quad (4.3)$$

In der untenstehenden 2D C# Implementierung ist zu sehen, wie zuerst ein Höhenwert für den aktuellen Punkt \vec{x} über das Skalarprodukt³ zwischen dem Gradienten und der relativen Position $\begin{pmatrix} tx \\ ty \end{pmatrix}$ $tx, ty \in [0 - 1]$ und anschließend über die bekannte Interpolation berechnet wird.

Gradient-Noise Implementierung C#

```

1 public float S(float x, float y) {
2     int floorX = Mathf.FloorToInt(x),
3         ceilX = Mathf.CeilToInt(x),
4         floorY = Mathf.FloorToInt(y),
5         ceilY = Mathf.CeilToInt(y);
6     float tx = x - floorX,
7         ty = y - floorY,
8         //Calc slopes
9         n00 = Vector2.Dot(LatticeFunc(floorX, floorY), new Vector2(tx, ty)),
10        n10 = Vector2.Dot(LatticeFunc(ceilX, floorY), new Vector2(tx-1, ty)),
11        n01 = Vector2.Dot(LatticeFunc(floorX, ceilY), new Vector2(tx, ty-1)),
12        n11 = Vector2.Dot(LatticeFunc(ceilX, ceilY), new Vector2(tx - 1, ty - 1));
13
14     return FadeFunction(1 - ty) *
15         (n00 * FadeFunction(1 - tx) + n10 * FadeFunction(tx))
16         +
17         FadeFunction(ty) *
18         (n01 * FadeFunction(1 - tx) + n11 * FadeFunction(tx));
19 }
```

4.4 Fractal-Noise

Die bisher behandelten Noise-Funktionen erzeugen zwar natürlich erscheinende Zufalls-werte, wenn man diese jedoch auf eine Heightmap überträgt wird deutlich, dass es ihr an Details fehlt um natürlich zu wirken.

³ engl. Dot-Product

4 Noise

Um den Detailgrad der Noise-Funktion beliebig zu erhöhen, wird die Noise-Funktion mit einer gestauchten, in der Amplitude verringerten, Version ihrer selbst addiert. Dieses Verfahren lässt sich beliebig oft anwenden, was einen beliebig hohen Detailgrad erlaubt.

In [Sau88] wird daher folgende Formel definiert:

$$\mathbb{H}(\vec{x}) = \sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} S(r^k \vec{x}). \quad (4.4)$$

Wobei $H = 2 - D$ der Hurst-Exponent ist[JS06] und $D = -\frac{\log(\frac{1}{k_1})}{\log(r)}$ [fra] die fraktale Dimension.

Durch die Anpassung dieser Parameter lässt sich das Verhalten der Noise-Funktion gezielt steuern. Eine Implementierung in C# findet sich unten.

Fractal-Noise Implementierung C#

4.5 Ausblick: Simplex-Noise

Die Gleichung 4.5

$$a^2 + b^2 = c^2 \quad (4.5)$$

ist allseits bekannt und bedarf wohl keiner weiteren Erläuterung.

Auch nicht schlecht ist Abbildung 4.5.1. Aber überhaupt keinen Sinn macht Tabelle 4.1. Hieran sieht man den Vorteil des autoref-Befehls und das so Links erstellt werden.

Formen	Städte
Quadrat	Bunkenstedt
Dreieck	Laggenbeck
Kreis	Peine
Raute	Wakaluba

Tabelle 4.1: eine sinnlose Tabelle

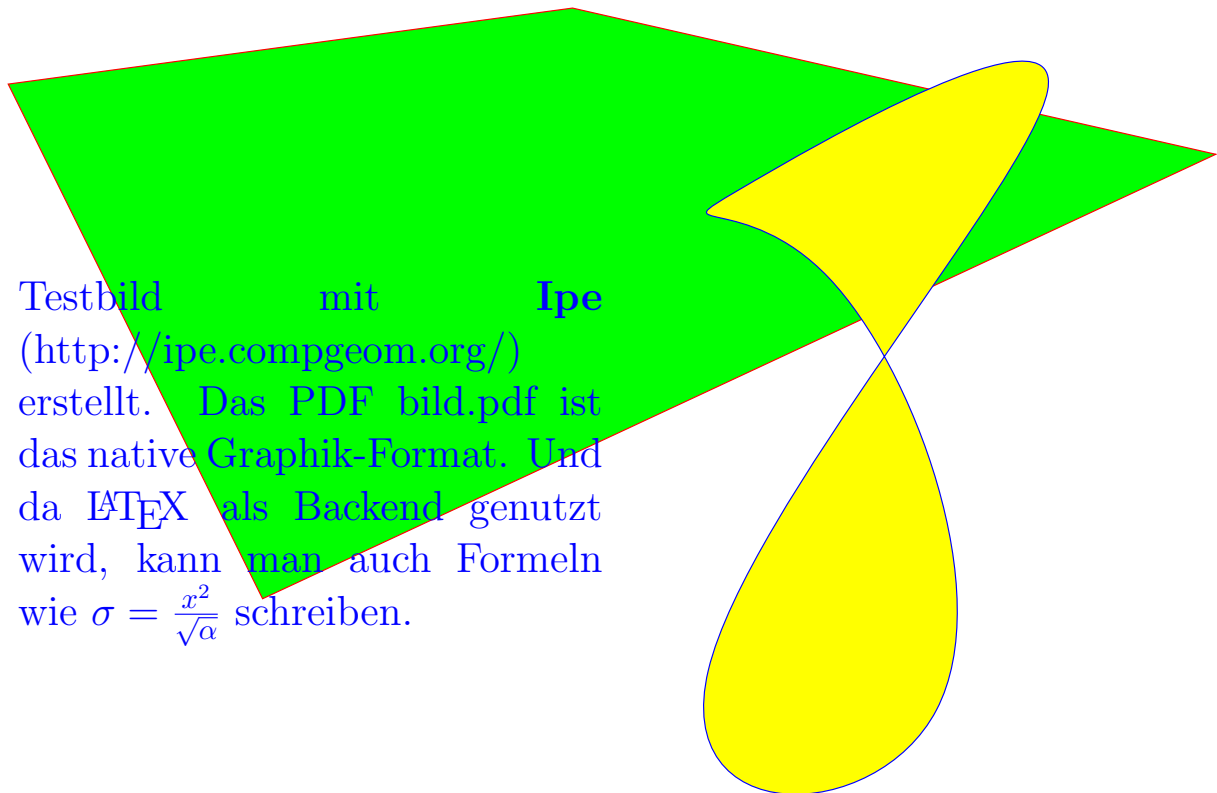
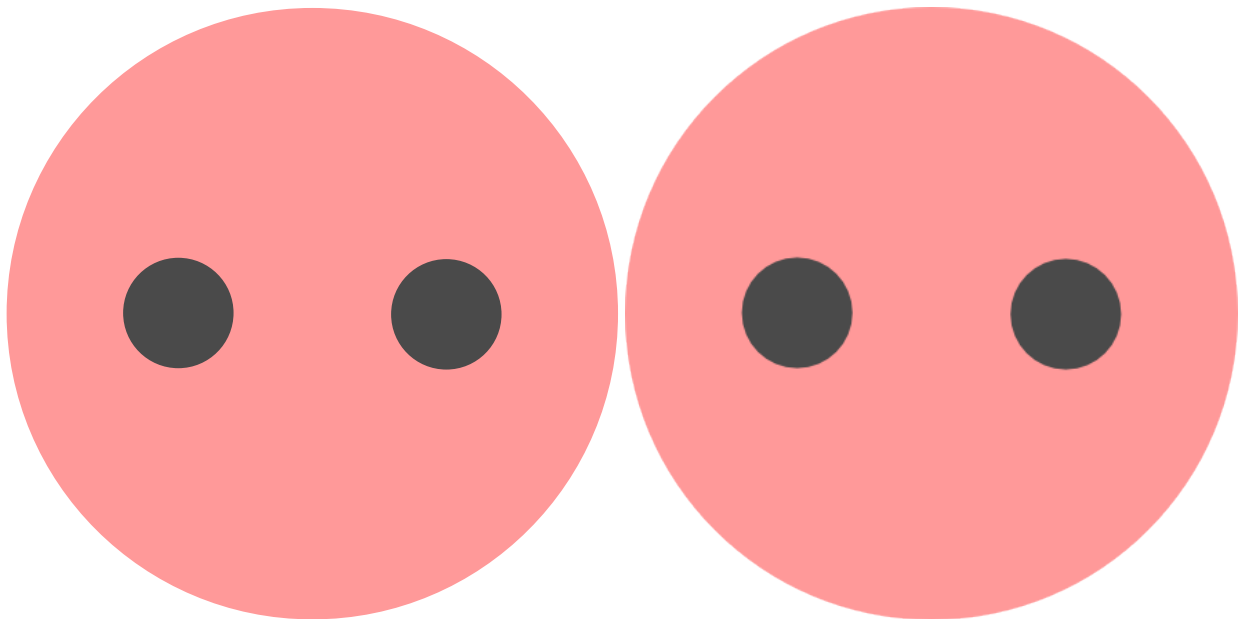


Abbildung 4.5.1: Test-Bild mit langer Bildunterschrift

4.5.1 jetzt geht es noch tiefer

Er hörte leise Schritte hinter sich. Das bedeutete nichts Gutes. Wer würde ihm schon folgen, spät in der Nacht und dazu noch in dieser engen Gasse mitten im übel beleumundeten Hafenviertel? Gerade jetzt, wo er das Ding seines Lebens gedreht hatte und mit der Beute verschwinden wollte! Hatte einer seiner zahllosen Kollegen dieselbe Idee gehabt, ihn beobachtet und abgewartet, um ihn nun um die Früchte seiner Arbeit zu erleichtern? Oder gehörten die Schritte hinter ihm zu einem der unzähligen Gesetzeshüter dieser Stadt, und die stählerne Acht um seine Handgelenke würde gleich zuschnappen? Er konnte die Aufforderung stehen zu bleiben schon hören. Gehetzt sah er sich um. Plötzlich erblickte er den schmalen Durchgang. Blitzartig drehte er sich nach rechts und verschwand zwischen den beiden Gebäuden. Beinahe wäre er dabei über den umgestürzten Mülleimer gefallen, der mitten im Weg lag. Er versuchte, sich in der Dunkelheit seinen Weg zu ertasten und erstarrte: Anscheinend gab es keinen anderen Ausweg aus diesem kleinen Hof als den Durchgang, durch den er gekommen war. Die Schritte wurden lauter und



(a) Ein Bild im PDF mit einer Größe von nur 1,1 kB **(b)** Das gleiche Bild als optimierte PNG-Datei mit einer Größe von 8,9 kB

Abbildung 4.5.2: Zwei Bilder werden mit dem \LaTeX -Paket subcaption nebeneinander angezeigt

lauter, er sah eine dunkle Gestalt um die Ecke biegen. Fieberhaft irrten seine Augen durch die nächtliche Dunkelheit und suchten einen Ausweg. War jetzt wirklich alles vorbei, waren alle Mühe und alle Vorbereitungen umsonst? Er presste sich ganz eng an die Wand hinter ihm und hoffte, der Verfolger würde ihn übersehen, als plötzlich neben ihm mit kaum wahrnehmbarem Quietschen eine Tür im nächtlichen Wind hin und her schwang. Könnte dieses der flehentlich herbeigesehnte Ausweg aus seinem Dilemma sein? Langsam bewegte er sich auf die offene Tür zu, immer dicht an die Mauer gepresst. Würde diese Tür seine Rettung werden?

Auch können Bilder in Bildern direkt angesprochen werden: Abbildung 4.5.2a und Abbildung 4.5.2b.

A Anhang

A.1 Quelltexte

cpu.c aus Linux 2.6.16

```

1  /* CPU control.
2   * (C) 2001, 2002, 2003, 2004 Rusty Russell
3   *
4   * This code is licenced under the GPL.
5   */
6  #include <linux/proc_fs.h>
7  #include <linux/smp.h>
8  #include <linux/init.h>
9  #include <linux/notifier.h>
10 #include <linux/sched.h>
11 #include <linux/unistd.h>
12 #include <linux/cpu.h>
13 #include <linux/module.h>
14 #include <linux/kthread.h>
15 #include <linux/stop_machine.h>
16 #include <asm/semaphore.h>
17
18 /* This protects CPUs going up and down... */
19 static DECLARE_MUTEX(cpucontrol);
20
21 static struct notifier_block *cpu_chain;
22
23 #ifdef CONFIG_HOTPLUG_CPU
24 static struct task_struct *lock_cpu_hotplug_owner;
25 static int lock_cpu_hotplug_depth;
26
27 static int __lock_cpu_hotplug(int interruptible)

```

```

28 {
29     int ret = 0;
30
31     if (lock_cpu_hotplug_owner != current) {
32         if (interruptible)
33             ret = down_interruptible(&cpucontrol);
34         else
35             down(&cpucontrol);
36     }
37
38     /*
39      * Set only if we succeed in locking
40      */
41     if (!ret) {
42         lock_cpu_hotplug_depth++;
43         lock_cpu_hotplug_owner = current;
44     }
45
46     return ret;
47 }
48
49 void lock_cpu_hotplug(void)
50 {
51     __lock_cpu_hotplug(0);
52 }
53 EXPORT_SYMBOL_GPL(lock_cpu_hotplug);
54
55 void unlock_cpu_hotplug(void)
56 {
57     if (--lock_cpu_hotplug_depth == 0) {
58         lock_cpu_hotplug_owner = NULL;
59         up(&cpucontrol);
60     }
61 }
62 EXPORT_SYMBOL_GPL(unlock_cpu_hotplug);
63
64 int lock_cpu_hotplug_interruptible(void)
65 {
66     return __lock_cpu_hotplug(1);
67 }
68 EXPORT_SYMBOL_GPL(lock_cpu_hotplug_interruptible);
69 #endif /* CONFIG_HOTPLUG_CPU */
70
71 /* Need to know about CPUs going up/down? */
72 int register_cpu_notifier(struct notifier_block *nb)
73 {
74     int ret;
75
76     if ((ret = lock_cpu_hotplug_interruptible()) != 0)
77         return ret;
78     ret = notifier_chain_register(&cpu_chain, nb);
79     unlock_cpu_hotplug();
80     return ret;

```

```

81 }
82 EXPORT_SYMBOL(register_cpu_notifier);
83
84 void unregister_cpu_notifier(struct notifier_block *nb)
85 {
86     lock_cpu_hotplug();
87     notifier_chain_unregister(&cpu_chain, nb);
88     unlock_cpu_hotplug();
89 }
90 EXPORT_SYMBOL(unregister_cpu_notifier);
91
92 #ifdef CONFIG_HOTPLUG_CPU
93 static inline void check_for_tasks(int cpu)
94 {
95     struct task_struct *p;
96
97     write_lock_irq(&tasklist_lock);
98     for_each_process(p) {
99         if (task_cpu(p) == cpu &&
100             (!cputime_eq(p->utime, cputime_zero) ||
101              !cputime_eq(p->stime, cputime_zero)))
102             printk(KERN_WARNING "Task %s (pid=%d) is on cpu %d\n",
103                    state_u = 0, ld, u flags_u = 0, l, e, u | n",
104                    p->comm, p->pid, cpu, p->state, p->flags);
105     }
106     write_unlock_irq(&tasklist_lock);
107 }
108
109 /* Take this CPU down. */
110 static int take_cpu_down(void *unused)
111 {
112     int err;
113
114     /* Ensure this CPU doesn't handle any more interrupts. */
115     err = __cpu_disable();
116     if (err < 0)
117         return err;
118
119     /* Force idle task to run as soon as we yield: it should
120        immediately notice cpu is offline and die quickly. */
121     sched_idle_next();
122     return 0;
123 }
124
125 int cpu_down(unsigned int cpu)
126 {
127     int err;
128     struct task_struct *p;
129     cpumask_t old_allowed, tmp;
130
131     if ((err = lock_cpu_hotplug_interruptible()) != 0)
132         return err;
133

```

```

134 if (num_online_cpus() == 1) {
135     err = -EBUSY;
136     goto out;
137 }
138
139 if (!cpu_online(cpu)) {
140     err = -EINVAL;
141     goto out;
142 }
143
144 err = notifier_call_chain(&cpu_chain, CPU_DOWN_PREPARE,
145                          (void *) (long)cpu);
146 if (err == NOTIFY_BAD) {
147     printk("%s: attempt to take down CPU %d failed\n",
148            __FUNCTION__, cpu);
149     err = -EINVAL;
150     goto out;
151 }
152
153 /* Ensure that we are not runnable on dying cpu */
154 old_allowed = current->cpus_allowed;
155 tmp = CPU_MASK_ALL;
156 cpu_clear(cpu, tmp);
157 set_cpus_allowed(current, tmp);
158
159 p = __stop_machine_run(take_cpu_down, NULL, cpu);
160 if (IS_ERR(p)) {
161     /* CPU didn't die: tell everyone. Can't complain. */
162     if (notifier_call_chain(&cpu_chain, CPU_DOWN_FAILED,
163                            (void *) (long)cpu) == NOTIFY_BAD)
164         BUG();
165
166     err = PTR_ERR(p);
167     goto out_allowed;
168 }
169
170 if (cpu_online(cpu))
171     goto out_thread;
172
173 /* Wait for it to sleep (leaving idle task). */
174 while (!idle_cpu(cpu))
175     yield();
176
177 /* This actually kills the CPU. */
178 __cpu_die(cpu);
179
180 /* Move it here so it can run. */
181 kthread_bind(p, get_cpu());
182 put_cpu();
183
184 /* CPU is completely dead: tell everyone. Too late to complain. */
185 if (notifier_call_chain(&cpu_chain, CPU_DEAD, (void *) (long)cpu)
186     == NOTIFY_BAD)

```

```

187     BUG();
188
189     check_for_tasks(cpu);
190
191 out_thread:
192     err = kthread_stop(p);
193 out_allowed:
194     set_cpus_allowed(current, old_allowed);
195 out:
196     unlock_cpu_hotplug();
197     return err;
198 }
199 #endif /*CONFIG_HOTPLUG_CPU*/
200
201 int __devinit cpu_up(unsigned int cpu)
202 {
203     int ret;
204     void *hcpu = (void *) (long) cpu;
205
206     if ((ret = lock_cpu_hotplug_interruptible()) != 0)
207         return ret;
208
209     if (cpu_online(cpu) || !cpu_present(cpu)) {
210         ret = -EINVAL;
211         goto out;
212     }
213

```

```

214     ret = notifier_call_chain(&cpu_chain, CPU_UP_PREPARE, hcpu);
215     if (ret == NOTIFY_BAD) {
216         printk("%s: attempt to bring up CPU %u failed\n",
217             __FUNCTION__, cpu);
218         ret = -EINVAL;
219         goto out_notify;
220     }
221
222     /* Arch-specific enabling code. */
223     ret = __cpu_up(cpu);
224     if (ret != 0)
225         goto out_notify;
226     if (!cpu_online(cpu))
227         BUG();
228
229     /* Now call notifier in preparation. */
230     notifier_call_chain(&cpu_chain, CPU_ONLINE, hcpu);
231
232 out_notify:
233     if (ret != 0)
234         notifier_call_chain(&cpu_chain, CPU_UP_CANCELED, hcpu);
235 out:
236     unlock_cpu_hotplug();
237     return ret;
238 }

```

Literaturverzeichnis

- [AC] A.J. Crilly, R.A. Earnshaw H.: *Fractals and Chaos*
- [Bur08] Burger, Wilhelm: Gradientenbasierte Rauschfunktionen und Perlin Noise / School of Informatics, Communications and Media, Upper Austria University of Applied Sciences. Version: November 2008. <http://staff.fh-hagenberg.at/burger/>. Hagenberg, Austria, November 2008 (HGBTR08-02). – Forschungsbericht
- [DSE03] David S. Ebert, Darwyn Peachey Ken Perlin Steven W. F. Kenton Musgrave M. F. Kenton Musgrave: *Texturing & Modeling: A Procedural Approach, Third Edition*. 2003
- [FFC82] Fournier, Alain; Fussell, Don; Carpenter, Loren: Computer Rendering of Stochastic Models. In: *Commun. ACM* 25 (1982), Juni, Nr. 6, 371–384. <http://dx.doi.org/10.1145/358523.358553>. – DOI 10.1145/358523.358553. – ISSN 0001-0782
- [fra] *Fraktale Dimension*. https://www.wikiwand.com/de/Fraktale_Dimension#/.C3.84hnlichkeits-Dimension
- [Gus05] Gustavson, Stefan: *Simplex noise demystified*. <http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf>. Version: 2005
- [HH] H. Hauser, E. R.: *State of the Art in Procedural Noise Functions*. <https://www-sop.inria.fr/reves/Basilic/2010/LLCDDELPZ10/LLCDDELPZ10STARPNF.pdf>
- [JS06] Jens Schneider, Rudiger W. Tobias Boldte B. Tobias Boldte: *Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs*. http://www.cg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Research/Publications/2006/vmv06.pdf. Version: 2006

Literaturverzeichnis

- [K85] In: K, PERLIN: *An image synthesizer*. vol. 19. 1985, S. 287– 296
- [Per] In: Perlin, K.: *Improving noise*, S. 681–682
- [Rus] Russell, Eddie: *Eliminate Texture Confusion: Bump, Normal and Displacement Maps*.
<http://blog.digitaltutors.com/bump-normal-and-displacement-maps/>
- [Sau88] Saupe, D: Point evaluation of multi-variable random fractals. In: *Visualisierung in Mathematik und Naturwissenschaft, Bremer Computergraphik Tage*, 1988

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum

Unterschrift