

**Hauptseminar**  
**Was Sie schon immer über Spiele wissen wollten...**  
Wintersemester 2005/06

Ausarbeitung zum Vortrag  
**Prozedurale Labyrinth-Generierung**

Till Rathmann  
14.11.2005

Leiter: Dominic Schell  
Lehrstuhl für Programmiersysteme  
(Informatik 2)  
Universität Erlangen-Nürnberg

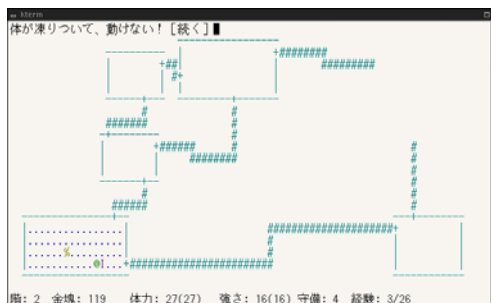
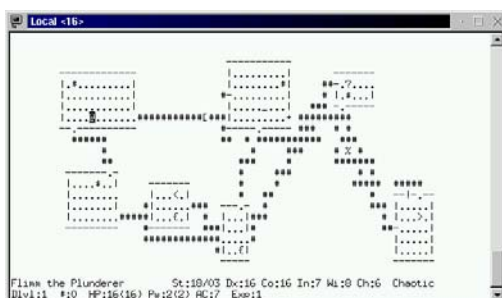
# Inhaltsverzeichnis

Einleitung .....	2
Zufälliges Rauschen .....	3
Einfache Labyrinth .....	3
Bessere Labyrinth .....	4
Spezielle Räume .....	5
Irreguläre Labyrinth .....	6
Alternativer Algorithmus .....	6
Implementierung .....	7
Literatur .....	8

## Einleitung

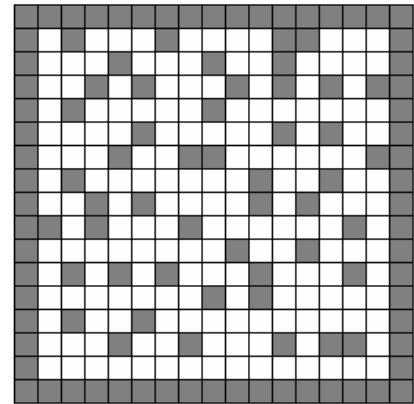
Zufällig generierte Dungeons wurden in Klassikern wie Nethack (unten links), Rogue (unten rechts) und Diablo (rechts) verwendet, um die Karte nicht bei jedem Spielstart gleich aussehen zu lassen. Vorteilhaft ist außerdem, dass bei dieser Technik keine Levels von Hand designed werden müssen. Deshalb haben wir uns entschlossen, in unserem Spielprojekt die Welt prozedural zu erzeugen. Das umfasst zum einen die Generierung der Landschaft (siehe separater Vortrag Terraingenerierung von Martin Passing). Zum anderen gibt es die Möglichkeit, die Map interessanter werden zu lassen, indem man in die Landschaft prozedural generierte Labyrinth einfügt. In unserem Spiel ist die Karte rechteckig und in viele Kacheln (Tiles) unterteilt. Aufgrund des Kachelkonzepts bestehen unsere Labyrinth ausschließlich aus senkrechten und waagrechten Wänden.

Im folgenden möchte ich Techniken vorstellen, mit denen sich Labyrinth automatisch erzeugen lassen.



## Zufälliges Rauschen

Die einfachste Methode, Maps zufällig zu erzeugen, besteht darin, den Typ von jedem einzelnen Block (Tile) per Zufallsgenerator zu bestimmen. Dazu werden einfach bestimmte Werte der Random-Funktion dem Typ „Wasser“, „Wiese“, „Fels“, usw. zugeordnet. Dass die so durch „random noise“ generierten Levels absolut chaotisch und unstrukturiert aussehen, ist leicht nachzuvollziehen. Ein Beispiel ist im Bild zu sehen. Um sinnvolle Levels zu erhalten, benötigt man Regeln, die eingehalten werden müssen. Alle Bereiche des Levels müssen von den Spielern erreichbar sein und abgetrennte Bereiche sollten nicht auftreten. Bei dieser Art von Levelgenerierung ist das natürlich nicht sichergestellt.



Pseudocode zur Generierung von Rauschen:

```
für jeden Block in der Map:  
    tile = random(0, NUM_TILES);
```

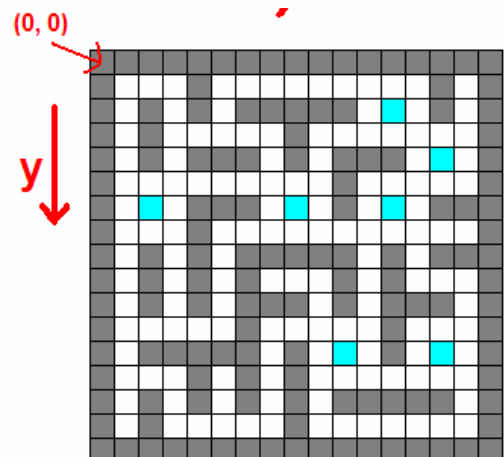
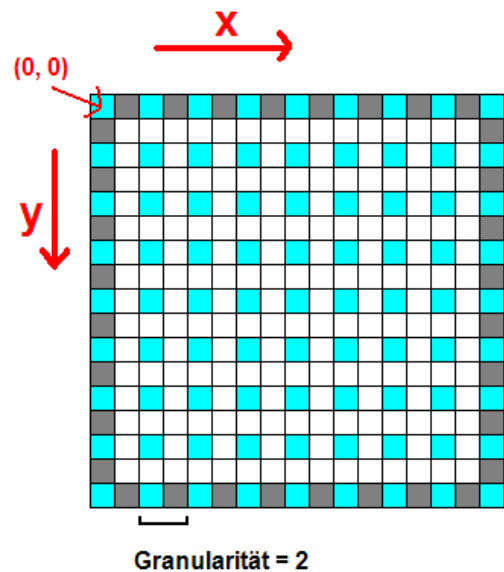
Mit Labyrinthen hat das zufällige Rauschen noch wenig zu tun. Dazu ist mehr Struktur nötig.

## Einfache Labyrinth

Die folgende Methode garantiert, dass kein Bereich der Map von den Spielern unerreichbar ist, also dass von jedem Punkt aus mindestens ein Weg zu jedem anderen Punkt existiert.

Um das Ergebnis nach Labyrinth aussehen zu lassen, wird probiert, von möglichen Wand-Startpositionen aus Wände zu zeichnen. Die Startpositionen der Wände werden per Zufallsgenerator berechnet, wobei ein gewisses Raster auf der Spielfläche berücksichtigt wird ( $x \bmod \text{granularity} = 0$  und  $y \bmod \text{granularity} = 0$ ). Sollen die Gänge z.B. ein Spielfeld breit sein, so wählt man als Granularität 2, für 3 Felder breite Gänge wählt man 4 als Granularität, usw.. Die Granularität bestimmt also den Abstand der Wände zueinander. Die obere Abbildung verdeutlicht dies (im Beispiel ist die Granularität 2 gewählt). Das untere Bild zeigt ein mögliches Labyrinth mit dieser Granularität.

Die Richtung (hoch, runter, links, rechts) und die Länge der Wand werden ebenfalls zufallsbestimmt, wobei bei letzterem bestimmte Minimal- und Maximalwerte vorgegeben werden. Befindet sich auf dem Startpunkt bereits eine Wand, dann wird auf diesem Feld keine neue Wand begonnen, wodurch

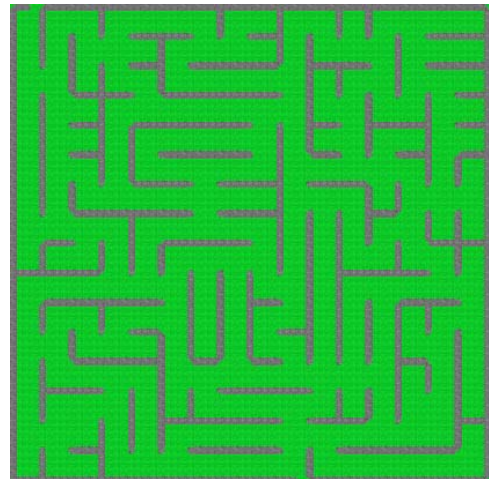
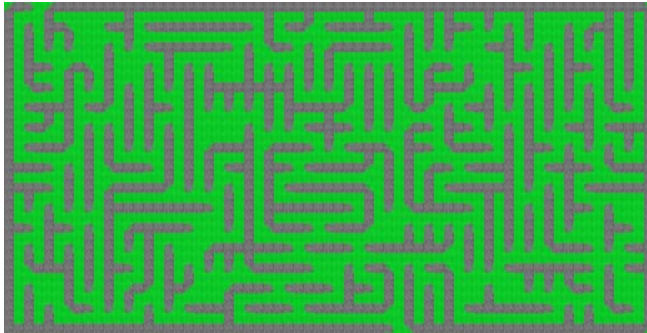


sichergestellt wird, dass keine Bereiche des Levels abgeschnitten und damit unerreichbar werden. Für das Zeichnen wird einfach über die Länge iteriert und an den entsprechenden Stellen in der Map das Tile auf „Wand“ gesetzt. Falls eine andere Wand oder sonstige Hindernisse erreicht werden, wird abgebrochen.

Auf diese Weise werden sehr viele Wände gezeichnet (bzw. probiert, zu zeichnen), wodurch man die Map nach und nach als Labyrinth-Charakter annimmt.

Pseudocode zum Erzeugen von Labyrinthen:

```
fillWithWalls(minLength, maxLength, granularity, numWalls) {
    numWalls-mal ausführen {
        // Startposition bestimmen
        x = granularity * random(0, (width-1) / granularity);
        y = granularity * random(0, (height-1) / granularity);
        // Richtung bestimmen
        dir = random(0, 3);
        // Länge bestimmen
        length = granularity * rand(minLength, maxLength) + 1;
        drawWall(x, y, dir, length);
    }
}
```



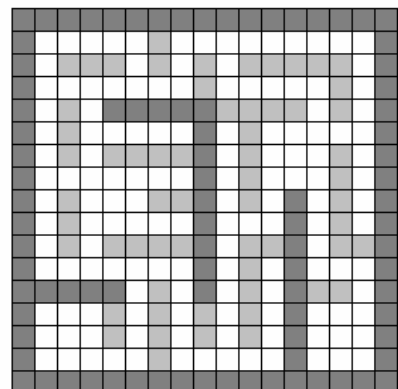
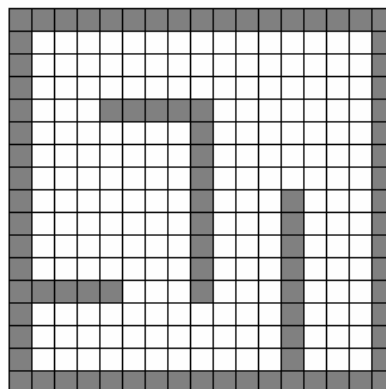
Das linke Bild zeigt ein Labyrinth mit Granularität 2, das rechte eins mit Granularität 4.

Das Ende der Fahnenstange ist aber noch nicht erreicht, wie wir im nächsten Abschnitt sehen werden.

## Bessere Labyrinthe

Die bisher erzeugten Labyrinthe sind relativ einfach und eintönig aufgebaut. Durch Kombinieren von Labyrinthen mit verschiedenen Granularitäten lassen sich die Dungeons komplexer und

abwechslungsreicher gestalten. Dies wird erreicht, indem der Wändezeichnen-Prozess zuerst mit einer hohen Granularität (also hohem Abstand der Wände) durchgeführt wird. Das unterteilt die Map in große Bereiche (siehe links). Danach lässt man den

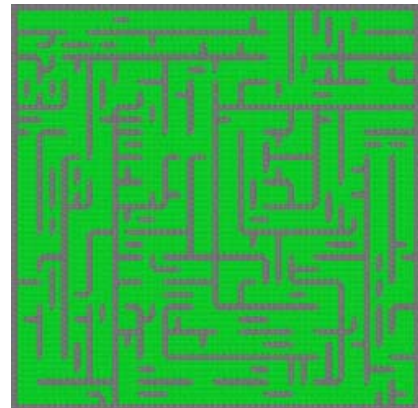


Algorithmus erneut über die Map laufen, allerdings mit kleinerer Granularität (siehe rechts). Führt man diesen Prozess ein paar mal durch (z.B. 4 mal, mit Granularität 16, 8, 4 und 2), so erhält man ein Labyrinth, das schwerer zu lösen ist und weniger eintönig aussieht. Die vorgeschalteten Durchläufe mit hoher Granularität bewirken nämlich, dass lange Wände entstehen, die den Weg versperren. Dadurch können im Labyrinth sehr lange Sackgassen enthalten sein, was die Komplexität zusätzlich erhöht. Durch Variieren der Anzahl der jeweils zu zeichnenden Wände, kann die Dichte der Labyrinth beeinflusst werden. Dies führt auch dazu, dass „Räume“ entstehen können, wenn Wände von Durchläufen kleinerer Granularität und geringer Dichte auf welche größerer Granularität treffen. Wegen der geringen Dichte werden dabei also längst nicht alle möglichen Wandstartpositionen benutzt, wodurch freie Bereiche entstehen, die aber wegen den vorherigen Durchläufen trotzdem gut mit Wänden umschlossen sind.

Codeausschnitt, mit dem das rechts zu sehende Labyrinth entstand:

```
// fillWithWalls(minLength, maxLength,
granularity, numWalls)
fillWithWalls(2, 4, 16, 200);
fillWithWalls(2, 4, 8, 400);
fillWithWalls(2, 4, 4, 1000);
fillWithWalls(1, 2, 2, 300);
```

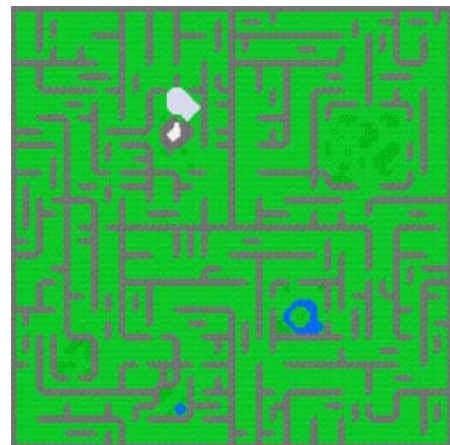
Diese Labyrinthe sind zwar schon recht brauchbar, aber es fehlt noch das gewisse Etwas. Darum kümmern wir uns im folgenden Abschnitt.



## Spezielle Räume

Wirklich interessante Dungeons entstehen erst, wenn zusätzlich zu den zufällig entstehenden Räumen spezielle vorgefertigte Räume eingefügt werden. Das können z.B. Kerkerzellen, Thronsäle, kleine Seen, usw. sein. Dazu werden einfach vorher festgelegte Räume (durch Arrays spezifiziert) in die Karte kopiert, bevor die Labyrinthgenerierung gestartet wird. Die Position, an die der Raum eingefügt werden soll, kann zufallsgeneriert werden. Allerdings müssen die Bereiche, die von Räumen belegt sind, als unveränderbar markiert werden, so dass diese nicht durch Labyrinthwände überschrieben werden.

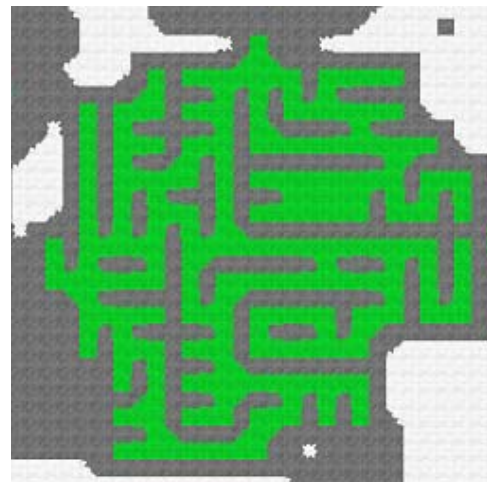
Alternativ zu den vordefinierten Räumen kann man auch den Terraingenerator von Martin Passing verwenden, um z.B. kleine Berge im Labyrinth zu erzeugen. Im Bild ist ein Labyrinth zu sehen, in das 5 verschiedene vordefinierte „Räume“ eingefügt wurden.



# Irreguläre Labyrinth

Um Labyrinth in Bergen in der Landschaft zu integrieren, ist es sinnvoll, dass diese keine rechteckige Form aufweisen. Um irreguläre Mazes zu erzeugen kann man folgendermaßen vorgehen:

Zuerst erstellt man eine Schablone (ein Array), die die Form des Labyrinths festlegt. Diese kann man z.B. aus den Terraindaten generieren, so dass das Labyrinth gut in einen Berg hinein passt. Zu beachten ist allerdings, dass die Eckpunkte der Kontur der Schablone auf geraden Koordinaten liegen müssen, damit der Labyrinthalgorithmus korrekt arbeiten kann. Eine zulässige Schablone wäre z.B. die folgende, mit der das zu sehende Labyrinth generiert wurde:

[illegible]

Danach werden die Ränder der Form gesucht und die entsprechenden Felder als Wand festgelegt. Ob ein Feld zur Kontur der Form gehört, erkennt man, wenn man die Schablonenwerte der 8 Nachbarpunkte verUNDet. Ist das Ergebnis eine 1, so liegt das Feld nicht am Rand. Bei 0 gehört das Feld zur Kontur. Nachdem also die Ränder mit Wänden beschrieben sind, kann der Labyrinthalgorithmus normal ausgeführt werden. Als einziger Unterschied ist zu beachten, dass als mögliche Startpositionen der Wände nur die in der Schablone markierten Felder in Frage kommen. Und natürlich müssen diese, genauso wie bei regulären Labyrinthen, durch 2, 4, 8, ... (je nach Granularität) teilbare Koordinaten aufweisen.

## Alternativer Algorithmus

Dass der oben beschriebene Algorithmus zur Labyrinthgenerierung nicht der einzige ist, ist nahe liegend. Als Beispiel für eine alternative Technik möchte ich zeigen, wie die Dungeons im Klassiker Rogue entstehen. Der dort verwendete Algorithmus geht den umgekehrten Weg im Vergleich zur bisher beschriebenen (und von uns verwendeten) Technik. Bei uns läuft die Generierung auf einer anfangs leeren Map und füllt diese nach und nach mit Wänden. Bei Rogue wird mit einer zuerst vollständig mit Fels gefüllter Map begonnen, in die nacheinander einzelne Räume und Gänge „gegraben“ werden. Das geschieht folgendermaßen:

- 1) einen Raum in die Map zeichnen
- 2) zufällig eine Raum- oder Gangwand auswählen



- 3) entscheiden, ob neuer Raum oder Gang gezeichnet werden soll
- 4) genug Platz für den neuen Raum/Gang?  
wenn ein Raum im Weg ist → Gang- / Raumlänge reduzieren, so dass eine Verbindung entsteht
- 5) Raum / Gang zeichnen
- 6) ab Schritt 2 wiederholen, bis genug Räume und Gänge existieren

Die dadurch entstehenden Maps sind mehr dungeonhaft und haben weniger mit Labyrinthen zu tun.

Im folgenden Abschnitt komme ich wieder auf unseren Algorithmus zurück und beschreibe das Interface der Klasse `Maze` und Besonderheiten, die beachtet werden müssen.

## Implementierung

Damit wir in unserem Spiel prozedurale Labyrinth durchwandern können, habe ich den beschriebenen Algorithmus in der Klasse `Maze` implementiert. Diese kann dann von der Klasse `Terrain` oder `Map` verwendet werden, um die Labyrinth in die Landschaft zu integrieren. Per

```
generateMaze(int width, int height)
```

wird das Labyrinth erzeugt. Die Breite und Höhe muss ungerade sein. Gleiches gilt für

```
generateIrregularMaze(int width, int height, const bool* shape)
```

womit irreguläre Labyrinth erzeugt werden. Die Schablone wird als Boolean-Array übergeben.

Mit

```
getTile(int x, int y)
```

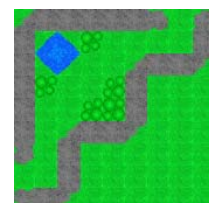
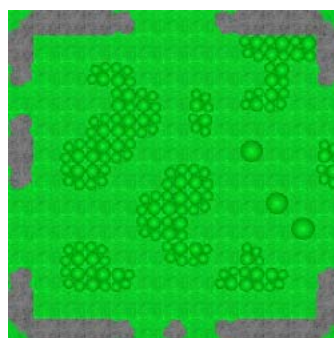
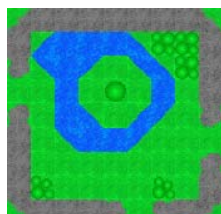
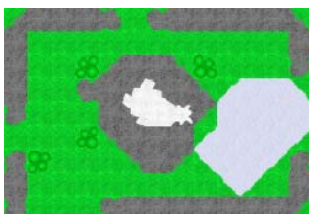
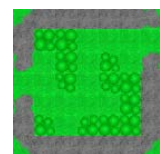
können die Labyrinth-Daten dann ausgelesen werden. -1 wird zurückgeliefert, falls das Labyrinth irregulär ist und das Feld in der Schablone nicht markiert ist.

Die Funktionen `getWidth()` und `getHeight()` liefern die Ausmaße des Maze zurück. Die Funktion

```
setTile(int x, int y, int tileType)
```

ist enthalten, damit der Benutzer der Klasse die Ein- bzw. Ausgänge setzen kann. Dabei ist allerdings zu beachten, dass diese nur an Positionen mit ungerader x- oder y-Koordinate liegen dürfen. Damit ist sichergestellt, dass direkt hinter dem Eingang keine Wände liegen können, die natürlich ein Eintreten verhindern würden.

Ich habe 5 spezielle Räume (siehe unten) in die `Maze`-Klasse integriert, die automatisch in das Labyrinth eingefügt werden, wobei die Ausmaße des Maze darüber entscheiden, wie viele Räume eingefügt werden. Die Räume werden außerdem rotiert (0°, 90°, 180° oder 270°), um etwas mehr Abwechslungsreichtum vorzutäuschen ☺



## Literatur

- Joshua Tippetts: Mazes and Labyrinths — An Introduction to Random Levels  
[http://members.gamedev.net/vertexnormal/tutorial\\_randlev1.html](http://members.gamedev.net/vertexnormal/tutorial_randlev1.html)
- Martin Passing: Prozedurale Terraingenerierung  
<http://www2.informatik.uni-erlangen.de/Lehre/WS200506/GameAlgHS>
- Damjan Jovanovic: Roguelike Development FAQ  
<http://www.roguelikedevelopment.org/development/FAQ.php>